



redhat®

RHD361

Red Hat Enterprise Linux Kernel Internals

RHD361-RHEL5-en-3-20091009

RED HAT TRAINING

Table of Contents

RH361 - Red Hat Enterprise Linux Kernel Internals

Red Hat Enterprise Linux Kernel Internals

Copyright	xi
Welcome	xii
Red Hat Enterprise Linux	xiii
Red Hat Enterprise Linux Variants	xiv
Red Hat Subscription Model	xv
Contacting Technical Support	xvi
Red Hat Network	xvii
Red Hat Services and Products	xviii
Fedora and EPEL	xix
RHD361 Objectives	xx
Audience and Prerequisites	xxi

Lecture 1 - Working with the Developer Community

Objectives	
Community Linux Kernel Development	2
Why Contribute Kernel Code Upstream?	3
Licensing	4
Copyright	5
Submitted Work	6
The Kernel Development Process	7
Creating Patches for the Merge Window	8
Staging Trees	9
End of Lecture 1	10
Lab 1: The Kernel Developer Community	
Lab 1.1: Following the Linux Kernel Developer Community	11

Lecture 2 - User Mode and Kernel Mode

Objectives	
The Linux Kernel - An Overview	14
The Role of the Kernel	15
Kernel Contexts	16
Four Milliseconds in the Life of the Kernel	17
System Ring Levels	18
Kernel Mode	19
User Mode	20
Mode Switching Example: System Calls	21
x86 System Call Interface	22
x86 System Call Interface (cont.)	23
Mode Switching Example: IRQ Event	24
Kernel Mode Linux	25
End of Lecture 2	26

Lab 2: User Mode and Kernel Mode

Lab 2.1: Exploring Kernel Execution Contexts	27
Lab 2.2: Exploring Kernel Transitions	28

Lecture 3 - Kernel Compilation and Tools

Objectives	
Kernel Packages	32
Kernel Version	33
Kernel Documentation	34
Kernel Source Layout	35
Kernel Source Layout (cont.)	36
Recompiling the Red Hat Kernel	37
Install Kernel Development Packages	38
Kernel Source Package	39
Preparing Source Code for Compilation	40
Customizing Kernel Name (Optional)	41
Choosing Compilation Options	42
Compiling the Kernel and Modules	43
Installing the Kernel Modules	44
Installing the Compiled Kernel and Related Files	45
Kernel Application Binary Interface (kABI)	47
cscope	48
LXR	49
git	50
git Documentation	51
End of Lecture 3	52
Lab 3: Compiling the Kernel	
Lab 3.1: Compiling the Red Hat Kernel	53
Lab 3.2: Searching Kernel Source with cscope	54
Optional Lab 3.3: Introducing LXR	55
Challenge Lab 3.4: Managing Revisions with git	56

Lecture 4 - Modules

Objectives	
Kernel Modules	62
Kernel Module Utilities	63
Mapping Modules to Attached Devices	65
Kernel Module Essentials	66
modinfo Macros	67
<code>printk()</code>	69
<code>/proc/kmsg</code> and klogd	70
<code>printk()</code> Loglevels	71
Rate Limiting <code>printk()</code>	73
Putting It All Together: A Simple Module	74
Compiling a Module	75
Integrating A New Module with the Kernel	76
Makefile and Kconfig	77
Module Parameters	78

Example: Module with Parameter	79
End of Lecture 4	81
Lab 4: Modules	
Lab 4.1: Building and Loading a Simple Module	82
Lab 4.2: Deploying a Kernel Module	83
Lab 4.3: Module Initialization and Exit Functions	84
Lab 4.4: Implementing Parameters in a Kernel Module	85

Lecture 5 - Kernel API Overview

Objectives	
Multitasking, Stacks, and Task-Descriptors	93
Contents of a Program's Stack	94
Kernel Mode Switch and the Stack	95
Task Structures	96
What Is a Process?	97
thread_info Structure	98
task_struct: Process Identifiers	99
task_struct: Process State	100
task_struct: Scheduling Information	102
Doubly Linked Lists	103
Doubly Linked Lists: Manipulation	104
Doubly Linked Lists: Iteration	105
Doubly Linked Lists: Processes	106
task_struct: Related Processes	107
task_struct: Statistics	108
Allocating Kernel Memory: kmalloc()	109
Memory Cache Optimizations: Branch Prediction	110
Memory Cache Optimizations: Binding Structures	111
Generating Kernel Errors	112
End of Lecture 5	113
Lab 5: Task Structures	
Lab 5.1: Querying Process Task Structures	114
Lab 5.2: Managing Linked Lists	115

Lecture 6 - Synchronization

Objectives	
Critical Sections	119
Mutual Exclusion Devices	120
Linux Mutex Toolbox	121
Atomic Bit Operations	122
Atomic Integers	123
Spinlocks	124
Spinlocks and Local Interrupts	125
Read-Write Spinlocks	126
Mutexes	127
Semaphores	128
Spinlock/Mutex Example	129
Alternatives to Locking	130

Sequential Locks	131
Read-Copy-Update (RCU)	133
Linux RCU Implementation	134
Per-CPU Variables	135
Completions	136
The Big Kernel Lock	137
End of Lecture 6	138
Lab 6: Synchronization	
Lab 6.1: Safely Querying Process Task Structures	139

Lecture 7 - Kernel Debugging 1: Tools and Techniques

Objectives	
Debugging Preparations	142
kernel-debuginfo Warnings	143
Kernel vs. User Space	144
Live vs. Postmortem Debugging	145
Crashes vs. Hangs	146
Debugging Device Drivers	147
User Space Debugging Tools	148
/proc Kernel Information	149
kernel.panic Tunable and Kernel Crashes	150
/sys Filesystem	151
debugfs Filesystem	152
Printing from the Kernel	153
Kernel Oops Messages	154
SysRq Mechanism	155
sosreport	156
The crash Tool	157
crash Requirements	158
crash Installation	159
crash Invocation	160
crash Invocation Output	161
crash Help	162
crash Command Input	163
crash Command Output	164
crash Command Overview	165
crash Default Context	166
End of Lecture 7	167
Lab 7: Kernel Debugging	
Lab 7.1: Preparing a System for Kernel Debugging	168
Lab 7.2: Gathering System Information	169
Lab 7.3: Controlling Processes with the Magic SysRq Key	170

Lecture 8 - Interrupts

Objectives	
Interrupts	175
Nature of Interrupts	176
Types of Interrupts	177

Interrupt Specific Hardware	178
Interrupt Descriptor Table (IDT)	179
IDT Initialization	180
IDT Initialization Functions	181
Exception Handling	182
Asynchronous Interrupt Handling	183
Interrupt Handler Considerations	184
<code>irq_desc</code> Structure	185
<code>irqaction</code> Structure	186
Interrupt Handler Registration	187
Performing Deferred Work	188
Softirqs	189
Using Softirqs	190
Tasklets	191
Using Tasklets	192
Work Queues	193
Work Queue Data Structures	194
Using Work Queues	195
End of Lecture 8	196
Lab 8: Interrupts	
Lab 8.1: Introduction to Interrupt Handlers	197
Lab 8.2: Bottom-Half Handling with Tasklets	198

Lecture 9 - Device Driver Overview

Objectives	
Device Drivers	201
Device Types	202
Device Nodes	203
Creating a Device Node	204
Dynamic Loading of Driver Modules	205
Major and Minor Numbers	206
Dynamic Major and Minor Numbers	207
Dynamically Created Device Nodes	208
Dynamically Created Device Nodes Made Easy	209
Device Driver Essentials	210
Character Device Registration	211
Device Driver File Operations	212
Driver Methods	213
The <code>file</code> Structure	215
The <code>inode</code> Structure	216
The <code>open</code> and <code>release</code> Methods	217
The <code>read</code> and <code>write</code> Methods	218
Module Usage Count	219
Simple Character Driver Example (1 of 3)	220
Simple Character Driver Example (2 of 3)	221
Simple Character Driver Example (3 of 3)	222
End of Lecture 9	223
Lab 9: Character Device Drivers	
Lab 9.1: Writing a Read-Only Character Device Driver	224

Lecture 10 - Memory Management

Objectives

Virtual Memory and Paging	228
x86 Memory Architecture	229
Memory Segmentation in Linux	230
x86 Segmentation	231
x86 Segmentation in Linux	232
Memory Paging	234
Page Tables	235
Mapping Virtual Addresses (x86)	236
Mapping Virtual Addresses (x86-64)	237
Memory Zones	238
Arranging the Virtual Address Space	239
ZONE_NORMAL	241
ZONE_HIGHMEM	242
ZONE_DMA	243
Kernel Memory Allocation	244
Memory Management	245
Buddy Allocator	246
Requesting and Releasing Page Frames	247
Slab Allocator	248
Slab Allocator (cont.)	249
Non-Contiguous Memory Area Management	250
Non-Contiguous Memory Area Management (cont.)	251
Memory Flags: gfp_mask	252
__get_free_pages()	253
kmalloc()	254
vmalloc()	255
End of Lecture 10	256
Lab 10: Memory Management	
Lab 10.1: Allocating Kernel Dynamic Memory	257
Lab 10.2: Using a Custom Slab Allocator	258

Lecture 11 - Processes

Objectives	
Creating Processes	261
Sharing Resources	262
do_fork()	263
Process Memory Maps	264
Memory Areas	265
vm_flags	266
pmap	267
Kernel Threads	268
Process 0	269
Process 0 (cont.)	270
Destroying Processes	271

Context Switches	272
When Does Context Switching Occur?	273
When Is <code>need_resched</code> Set?	274
When Is <code>schedule()</code> Called?	275
Kernel Preemption	276
End of Lecture 11	277
Lab 11: Processes	
Lab 11.1: Kernel Threads	278
Lab 11.2: Process Creation	279

Lecture 12 - The Scheduler

Objectives	
Priority	283
Priority for Normal Processes	284
Priority for Real-Time Processes	285
Time Slices	286
The O(1) Scheduler: Run Queues	287
The O(1) Scheduler: Priority Arrays	289
The O(1) Scheduler: How it works	290
The O(1) Scheduler (cont.)	291
Wait Queues	292
The O(1) Scheduler: Load Balancing	293
The O(1) Scheduler: <code>load_balance()</code>	294
Problems with the O(1) Scheduler	295
O(1) Scheduler vs. CFS	296
Overview of CFS	297
Details of CFS	298
CFS Task Scheduling	299
CFS Scheduler Policies	300
CFS Scheduler Classes	301
CFS <code>fair_sched_class</code>	302
CFS Tuning	303
CFS Group Scheduling	304
<code>CONFIG_FAIR_GROUP_SCHED</code>	305
<code>CONFIG_FAIR_CGROUP_SCHED</code>	306
End of Lecture 12	307
Lab 12: The Scheduler	
Lab 12.1: Scheduling Standard Priority Processes	308
Lab 12.2: Scheduling Real-Time Priority Processes	309

Lecture 13 - Kernel Timing

Objectives	
The Need for Timing	313
Timing Hardware	314
Timing Source Selection	315
Wall/Real Time: <code>xtime</code>	316
Wall Clock System Calls	317
Kernel Ticks: <code>jiffies</code>	318

Software Timers	319
POSIX Timers	320
Interval Timers and <code>alarm()</code>	321
High-Resolution Timers	322
Timer Interrupt Handler	323
<code>TIMER_SOFTIRQ</code> Softirq	324
Delay Functions	325
End of Lecture 13	326
Lab 13: Kernel Timing	
Lab 13.1: Displaying Real Time	327
Lab 13.2: Creating an Internal Interval Timer	328

Lecture 14 - SystemTap

Objectives	
Introduction to SystemTap	331
SystemTap's Main Components	332
Monitoring the Kernel with SystemTap	333
The stap Command	334
Flow of Data in SystemTap	335
Common Tapset Probe Points	336
SystemTap Script Examples	337
End of Lecture 14	338
Lab 14: Installing SystemTap and Using Tapsets	
Lab 14.1: Monitoring Context Switches with SystemTap	339

Lecture 15 - System and Kernel Initialization

Objectives	
Boot Sequence Overview	342
BIOS Initialization	343
Bootloader	344
Starting the Boot Process: GRUB	345
Bootloader Components	346
The Chicken/Egg Module Problem and the Initial RAM Disk	347
GRUB and <code>grub.conf</code>	349
Kernel Initialization Overview	350
<code>__init</code> and <code>__initdata</code>	351
Initialization Subsections and Ordering	352
Kernel Initialization	354
<code>init/main.c: start_kernel()</code>	355
<code>init/main.c: rest_init()</code>	356
<code>init/main.c: init()</code>	357
<code>init/main.c: do_basic_setup()</code>	358
<code>init/main.c: init_post()</code>	359
init Initialization	360
Run Levels	361
End of Lecture 15	362
Lab 15: Managing Startup	
Lab 15.1: Exploring an Initial RAM Disk	363

Lecture 16 - Kernel Debugging 2: Crash Dumps

Objectives	
Introduction to Crash Dumps	368
Netdump/Diskdump Challenges	369
Kdump	370
Kdump Solution	371
Kexec	372
Relocatable Kernel	373
In-place Kernel Decompression	374
Starting Kdump	375
Kdump Initrd Image	376
Configuring Kdump	377
Kdump Core Dumps to the Local System	378
Kdump Core Dumps to NFS Mount Points	379
Kdump Core Dumps to SSH Servers	380
Dump File Size	381
Customizing the Dump Capture Method: makedumpfile	382
Dump Filtering	383
Dump Compression	384
Future Challenges	385
End of Lecture 16	386
Lab 16: Kernel Crash Dumps	
Lab 16.1: Implementing Kdump/Kexec	387

Lecture 17 - Red Hat Enterprise Linux Realtime Kernel

Objectives	
Realtime (RT) Linux	391
Benefits of a Realtime Kernel	392
Response Time Comparisons	393
Wake-Up Response Time Example	394
Changes in the Kernel	395
Changes in the C Library	396
RT Measurement Tools	397
RT Tuning Tools	398
RT Tuning Methods	399
Loading the RT Kernel	400
End of Lecture 17	401



Introduction

Red Hat Enterprise Linux Kernel Internals

For use only by a student enrolled in a Red Hat training course taught by Red Hat, Inc. or a Red Hat Certified Training Partner. No part of this publication may be photocopied, duplicated, stored in a retrieval system, or otherwise reproduced without prior written consent of Red Hat, Inc. If you believe Red Hat training materials are being improperly used, copied, or distributed please email <training@redhat.com> or phone toll-free (USA) +1 (866) 626 2994 or +1 (919) 754 3700.



- The contents of this course and all its modules and related materials, including handouts to audience members, are Copyright © 2009 Red Hat, Inc.
- No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but not limited to, photocopy, photograph, magnetic, electronic or other record, without the prior written permission of Red Hat, Inc.
- This instructional program, including all material provided herein, is supplied without any guarantees from Red Hat, Inc. Red Hat, Inc. assumes no liability for damages or legal action arising from the use or misuse of contents or details contained herein.
- If you believe Red Hat training materials are being used, copied, or otherwise improperly distributed please email training@redhat.com or phone toll-free (USA) +1 866 626 2994 or +1 919 754 3700.



Please let us know if you need any special assistance while visiting our training facility.

Please introduce yourself to the rest of the class!

Welcome to Red Hat Training!

Welcome to this Red Hat training class! Please make yourself comfortable while you are here. If you have any questions about this class or the facility, or need special assistance while you are here, please feel free to ask the instructor or staff at the facility for assistance. Thank you for attending this course.

Telephone and network availability

Please only make telephone calls during breaks. Your instructor will direct you to the telephone to use. Network access and analog phone lines may be available; if so, your instructor will provide information about these facilities. Please turn pagers and cell phones to off or to silent or vibrate during class.

Restrooms

Your instructor will notify you of the location of restroom facilities and provide any access codes or keys which are required to use them.

Lunch and breaks

Your instructor will notify you of the areas to which you have access for lunch and for breaks.

In Case of Emergency

Please let us know if anything comes up that will prevent you from attending or completing the class this week.

Access

Each training facility has its own opening and closing times. Your instructor will provide you with this information.



- Enterprise-targeted Linux operating system
- Focused on mature open source technology
- Extended release cycle between major versions
 - With periodic minor releases during the cycle
 - Certified with leading OEM and ISV products
- All variants based on the same code
 - Certify once, run any application/anywhere/anytime
- Services provided on subscription basis

The Red Hat Enterprise Linux product family is designed specifically for organizations planning to use Linux in production settings. All products in the Red Hat Enterprise Linux family are built on the same software foundation, and maintain the highest level of ABI/API compatibility across releases and errata. Extensive support services are available: a one year support contract and Update Module entitlement to Red Hat Network are included with purchase. Various Service Level Agreements are available that may provide up to 24x7 coverage with a guaranteed one hour response time for Severity 1 issues. Support will be available for up to seven years after a particular major release.

Red Hat Enterprise Linux is released on a multi-year cycle between major releases. Minor updates to major releases are released roughly every six months during the lifecycle of the product. Systems certified on one minor update of a major release continue to be certified for future minor updates of the major release. A core set of shared libraries have APIs and ABIs which will be preserved between major releases. Many other shared libraries are provided, which have APIs and ABIs which are guaranteed within a major release (for all minor updates) but which are not guaranteed to be stable across major releases.

Red Hat Enterprise Linux is based on code developed by the open source community and adds performance enhancements, intensive testing, and certification on products produced by top independent software and hardware vendors such as Dell, IBM, Fujitsu, BEA, and Oracle. Red Hat Enterprise Linux provides a high degree of standardization through its support for five processor architectures (Intel x86-compatible, AMD64/Intel 64, Intel Itanium 2, IBM POWER, and IBM mainframe on System z). Furthermore, we support the 3000+ ISV certifications on Red Hat Enterprise Linux whether the RHEL operating system those applications are using is running on "bare metal", in a virtual machine, as a software appliance, or in the cloud using technologies such as Amazon EC2.



- Red Hat Enterprise Linux Advanced Platform
 - Unlimited server size and virtualization support
 - HA clusters and cluster file system
- Red Hat Enterprise Linux
 - Basic server solution for smaller non-mission-critical servers
 - Virtualization support included
- Red Hat Enterprise Linux Desktop
 - Productivity desktop environment
 - *Workstation* option adds tools for software and network service development
 - *Multi-OS* option for virtualization

Currently, on the x86 and x86-64 architectures, the product family includes:

Red Hat Enterprise Linux Advanced Platform: the most cost-effective server solution, this product includes support for the largest x86-compatible servers, unlimited virtualized guest operating systems, storage virtualization, high-availability application and guest fail-over clusters, and the highest levels of technical support.

Red Hat Enterprise Linux: the basic server solution, supporting servers with up to two CPU sockets and up to four virtualized guest operating systems.

Red Hat Enterprise Linux Desktop: a general-purpose client solution, offering desktop applications such as the OpenOffice.org office suite and Evolution mail client. Add-on options provide support for high-end technical and development workstations and for running multiple operating systems simultaneously through virtualization.

Two standard installation media kits are used to distribute variants of the operating system. Red Hat Enterprise Linux Advanced Platform and Red Hat Enterprise Linux are shipped on the *Server* media kit. Red Hat Enterprise Linux Desktop and its add-on options are shipped on the *Client* media kit. Media kits may be downloaded as ISO 9660 CD-ROM file system images from Red Hat Network or may be provided in a boxed set on DVD-ROMs.

Please visit <http://www.redhat.com/rhel/> for more information about the Red Hat Enterprise Linux product family. Other related products include realtime kernel support in Red Hat Enterprise MRG, the thin hypervisor node in Red Hat Enterprise Virtualization, and so on.



- Red Hat sells subscriptions that entitle systems to receive a set of services that support open source software
 - Red Hat Enterprise Linux and other Red Hat/JBoss solutions and applications
- Customers are charged an annual subscription fee per system
 - Subscriptions can be migrated as hardware is replaced
 - Can freely move between major revisions, up and down
 - Multi-year subscriptions are available
- A typical service subscription includes:
 - Software updates and upgrades through Red Hat Network
 - Technical support (web and phone)
 - Certifications, stable APIs/versions, and more

Red Hat doesn't exactly sell software. What we sell is *service* through support subscriptions.

Customers are charged an annual subscription fee per system. This subscription includes the ability to manage systems and download software and software updates through our Red Hat Network service; to obtain technical support (through the World-Wide Web or by telephone, with terms that vary depending on the exact subscription purchased), and extended software warranties and IP indemnification to protect the customer from service interruption due to software bugs or legal issues.

In turn, the subscription-based model gives customers more flexibility. Subscriptions are tied to a *service level*, not to a release version of a product; therefore, upgrades (and downgrades!) of software between major releases can be done on a customer's own schedule. Management of versions to match the requirements of third-party software vendors is simplified as well. Likewise, as hardware is replaced, the service entitlement which formerly belonged to a server being decommissioned may be freely moved to a replacement machine without requiring any assistance from Red Hat. Multi-year subscriptions are available as well to help customers better tie software replacement cycles to hardware refresh cycles.

Subscriptions are not just about access to software updates. They provide unlimited technical support; hardware and software certifications on tested configurations; guaranteed long-term stability of a major release's software versions and APIs; the flexibility to move entitlements between versions, machines, and in some cases processor architectures; and access to various options through Red Hat Network and add-on products for enhanced management capabilities.

This allows customers to reduce deployment risks. Red Hat can deliver new technology as it becomes available in major releases. But you can choose when and how to move to those releases, without needing to relicense to gain access to a newer version of the software. The subscription model helps reduce your financial risk by providing a road map of predictable IT costs (rather than suddenly having to buy licenses just because a new version has arrived). Finally, it allows us to reduce your technological risk by providing a stable environment tested with software and hardware important to the enterprise. Visit <http://www.redhat.com/rhel/benefits/> for more information about the subscription model.



- Collect information needed by technical support:
 - Define the problem
 - Gather background information
 - Gather relevant diagnostic information, if possible
 - Determine the severity level
- Contacting technical support by WWW:
 - <http://www.redhat.com/support/>
- Contacting technical support by phone:
 - See <http://www.redhat.com/support/policy/sla/contact/>
 - US/Canada: 888-GO-REDHAT (888-467-3342)

Information on the most important steps to take to ensure your support issue is resolved by Red Hat as quickly and efficiently as possible is available at <http://www.redhat.com/support/process/production/>. This is a brief summary of that information for your convenience. You may be able to resolve your problem without formal technical support by looking for your problem in Knowledgebase (<http://kbase.redhat.com/>).

Define the problem. Make certain that you can articulate the problem and its symptoms before you contact Red Hat. Be as specific as possible, and detail the steps you can use (if any) to reproduce the problem.

Gather background information. What version of our software are you running? Are you using the latest update? What steps led to the failure? Can the problem be recreated and what steps are required? Have any recent changes been made that could have triggered the issue? Were messages or other diagnostic messages issued? What *exactly* were they (exact wording may be critical)?

Gather relevant diagnostic information. Be ready to provide as much relevant information as possible; logs, core dumps, traces, the output of **sosreport**, etc. Technical Support can assist you in determining what is relevant.

Determine the Severity Level of your issue. Red Hat uses a four-level scale to indicate the criticality of issues; criteria may be found at http://www.redhat.com/support/policy/GSS_severity.html.

Red Hat Support may be contacted through a web form or by phone depending on your support level. Phone numbers and business hours for different regions vary; see <http://www.redhat.com/support/policy/sla/contact/> for exact details. When contacting us about an issue, please have the following information ready:

- Red Hat Customer Number
- Machine type/model
- Company name
- Contact name
- Preferred means of contact (phone/e-mail) and telephone number/e-mail address at which you can be reached
- Related product/version information
- Detailed description of the issue
- Severity Level of the issue in respect to your business needs



- A systems management platform providing lifecycle management of the operating system and applications
 - Installing and provisioning new systems
 - Updating systems
 - Managing configuration files
 - Monitoring performance
 - Redeploying systems for a new purpose
- "Hosted" and "Satellite" deployment architectures

Red Hat Network's modular service model allows you to pick and choose the features you need to manage your enterprise.

The basic Update service is provided as part of all Red Hat Enterprise Linux subscriptions. Through it, you can use Red Hat Network to easily download and install security patches and updates from an RHN server. All content is digitally signed by Red Hat for added security, so that you can ensure that packages actually came from us. The **yum** (or older **up2date**) utility automatically resolves dependencies to ensure the integrity of your system when you use it to initiate an update from the managed station itself. You can also log into a web interface on the RHN server to remotely add software or updates or remove undesired software packages, or set up automatic updates to allow systems to get all fixes immediately.

The add-on Management module allows you to organize systems into management groups and perform update or other management operations on all members of the group. You can also set up subaccounts in Red Hat Network which have access to machines in some of your groups but not others. Powerful search capabilities allow you to identify systems based on their packages or hardware characteristics, and you can also compare package profiles of two systems.

The Provisioning module makes it easier for you to deploy new systems or redeploy existing systems using predetermined profiles or through system cloning. You can use RHN to store, manage, and deploy configuration files as well as software package files. You can use tools to help write automated installation Kickstart configurations and apply them to selected systems. You can undo problematic changes through a roll-back feature. Management and Provisioning modules are included as part of a Red Hat Enterprise Linux Desktop subscription at no additional fee.

Monitoring module is only available with RHN Satellite, and allows you to set up to dozens of low-impact probes for each system and many applications (including Oracle, MySQL, BEA, and Apache) to track availability and performance.

RHN is initially deployed in a "hosted" model, where the central update server is located at a Red Hat facility and is contacted over the Internet using HTTP/SSL. To reduce bandwidth, you can site a RHN Proxy Server at your facility which caches packages requested by your systems. For maximum flexibility, you may use RHN Satellite, which places the RHN server at your site under your control; this can run disconnected from the Internet, or may be connected to the Internet to download update content from the official hosted RHN servers to populate its service channels.



- Red Hat supports software products and services beyond Red Hat Enterprise Linux
 - JBoss Enterprise Middleware
 - Systems and Identity Management
 - Infrastructure products and distributed computing
 - Training, consulting, and extended support
- <http://www.redhat.com/products/>

Red Hat offers a number of additional open source application products and operating system enhancements which may be added to the standard Red Hat Enterprise Linux operating system. As with Red Hat Enterprise Linux, Red Hat provides a range of maintenance and support services for these add-on products. Installation media and software updates are provided through the same Red Hat Network interface used to manage Red Hat Enterprise Linux systems.

For additional information, see the following web pages:

- General product information: <http://www.redhat.com/products/>
- Red Hat Solutions Guide: <http://www.redhat.com/solutions/guide/>



- Open source projects sponsored by Red Hat
- Fedora distribution is focused on latest open source technology
 - Rapid six month release cycle
 - Available as free download from the Internet
- EPEL provides add-on software for Red Hat Enterprise Linux
- Open, community-supported proving grounds for technologies which may be used in upcoming enterprise products
- Red Hat does not provide formal support

Fedora is a rapidly evolving, technology-driven Linux distribution with an open, highly scalable development and distribution model. It is sponsored by Red Hat but created by the Fedora Project, a partnership of free software community members from around the globe. It is designed to be a fully-operational, innovative operating system which also is an incubator and test bed for new technologies that may be used in later Red Hat enterprise products. The Fedora distribution is available for free download from the Internet.

The Fedora Project produces releases of Fedora on a short, roughly six month release cycle, to bring the latest innovations of open source technology to the community. This may make it attractive for power users and developers who want access to cutting-edge technology and can handle the risks of adopting rapidly changing new technology. Red Hat does not provide formal support for Fedora.

The Fedora Project also supports EPEL, Extra Packages for Enterprise Linux. EPEL is a volunteer-based community effort to create a repository of high-quality add-on packages which can be used with Red Hat Enterprise Linux and compatible derivatives. It accepts legally-unencumbered free and open source software which does not conflict with packages in Red Hat Enterprise Linux or Red Hat add-on products. EPEL packages are built for a particular major release of Red Hat Enterprise Linux and will be updated by EPEL for the standard support lifetime of that major release.

Red Hat does not provide commercial support or service level agreements for EPEL packages. While not supported officially by Red Hat, EPEL provides a useful way to reduce support costs for unsupported packages which your enterprise wishes to use with Red Hat Enterprise Linux. EPEL allows you to distribute support work you would need to do by yourself across other organizations which share your desire to use this open source software in RHEL. The software packages themselves go through the same review process as Fedora packages, meaning that experienced Linux developers have examined the packages for issues. As EPEL does not replace or conflict with software packages shipped in RHEL, you can use EPEL with confidence that it will not cause problems with your normal software packages.

For developers who wish to see their open source software become part of Red Hat Enterprise Linux, often a first stage is to sponsor it in EPEL so that RHEL users have the opportunity to use it, and so experience is gained with managing the package for a Red Hat distribution.

Visit <http://fedoraproject.org/> for more information about the Fedora Project.

Visit <http://fedoraproject.org/wiki/EPEL/> for more information about EPEL.



- Introduce kernel development tools
 - Kernel compilation and installation
 - Source code search tools
 - Debugging tools and techniques
- Examine the operation of the Linux kernel
 - Interaction between user and kernel mode
 - Developing kernel modules
 - Synchronization tools and techniques
 - Interrupts
 - High-level device driver API
 - Memory management
 - Process management and scheduling
 - Kernel timing
 - Kernel initialization
 - Working with the Linux community



- Audience:
 - System administrators wanting a deeper knowledge of the Red Hat Enterprise Linux kernel subsystems
 - Device driver developers
 - Performance tuners
- Prerequisites:
 - Proficiency with Linux command line tools and an editor
 - Proficiency with the C programming language
 - Understand how to compile and link a C program and construct a Makefile



Lecture 1

Working with the Developer Community

Upon completion of this unit, you should be able to:

- Give reasons for contributing open-source code
- Understand legal requirements for contributions
- Understand the kernel release cycle



- Linux development is very active!
- Some numbers:
 - Over 9 million lines of code
 - More than 1000 active developers
 - Approximately 10,000 patches in a recent release
 - Thousands of lines added, removed, and changed each *day*
- Community-supported procedures manage such rapid growth
 - Getting started with LKML and Kernelnewbies

Note: The numbers stated above were derived from: <http://www.linuxfoundation.org/publications/linuxkerneldevelopment.php>.

One way to get started participating in the Linux kernel development community is to subscribe to the LKML - Linux-Kernel Mailing List. It is a high volume list (sometimes over 500 messages a day) where kernel developers interact with each other about technical and administrative issues concerning kernel development. You can find out how to subscribe to the list by reviewing the LKML FAQ found at <http://www.tux.org/lkml>. Another useful site is <http://lkml.org> which is an archive of previous LKML correspondence.

Another way to learn more about getting started with kernel development is to visit the <http://kernelnewbies.org> site. The site has useful kernel programming FAQs information. Kernelnewbies also has a mailing list and an IRC channel where you can lurk or post questions.



- Often called the *mainline* kernel, it is the version maintained by Linus Torvalds and used as a base for the Red Hat distribution
- Some reasons for getting your code into the mainline kernel:
 - Ability to influence the direction of kernel development
 - Automatic inclusion in future releases
 - Lower maintenance: developers must fix any of your code that breaks as a result of API changes
 - A rigorous review process at submission time can improve code
 - An increased likelihood of the code being improved later by other developers and the community
 - Reduce risk of 3rd party code becoming de facto standard and having to migrate your users to it
 - An investment in the continued success of Linux

While submitting code upstream for inclusion into the mainline kernel can greatly reduce the maintenance of said code, total absence of the original developer after inclusion is not looked favorably upon by the developer community. Some continued responsibility for the code is helpful to the community and the long-term viability of the code.



- The kernel as a whole is covered by GPLv2
- Code is contributed to the kernel under a number of licenses, but it must be compatible with:
 - GPLv2 (with, optionally, language allowing distribution under later versions of the GPL), or
 - the three-clause BSD license
- Submissions not covered by these licenses will not be accepted

GPLv2: <http://www.gnu.org/licenses/gpl-2.0.html>

"3-clause" BSD: <http://www.opensource.org/licenses/bsd-license.php>



- Copyright assignments are not required or requested for kernel code submission
- All code accepted into the mainline kernel retains its original ownership
- There are thousands of kernel owners!
- A notable side effect:
 - Changing the kernel's license to the satisfaction of all owners is practically impossible
 - As a result, no GPLv3 for the kernel's near future



- All submitted works of code must be free:
 - No anonymous/pseudonymous submissions
 - Contributors must "sign off" that their code can be distributed with the kernel under the GPL
- Final legal note: IANAL!
 - ...and neither are most kernel mailing list posters providing advice
 - Always consult a lawyer who understands this field

In the kernel source tree there is a file called `Documentation/CodingStyle` which contains a document entitled "Linux kernel coding style." It describes the preferred coding style for the Linux kernel and includes topics such as:

- Indentation
- Breaking long lines and strings
- Placing spaces and braces
- Identifier naming

and other programming style-related topics. You will definitely want to review this document before you submit code for inclusion into the upstream kernel.



- Start with a stable kernel release (released about every 2-3 months)
- Linus opens the *merge window* for the next release
 - Sufficiently stable code is merged into the mainline kernel
 - Approximately 1000 changes per day
 - 2-week window
- Linus closes the merge window, creates the first `-rc` release
 - No new features will be merged, stabilization begins
 - 6-10 weeks of patching
 - A new `-rc` release approximately every week (e.g. `2.6.X-rc1`, `2.6.X-rc2`, ...)
- A new stable version of the kernel is released
- Occasional stable release updates
 - Must already be accepted into mainline of next development kernel
 - Rules for stable kernel updates: `Documentation/stable_kernel_rules.txt`
 - Version format: `2.6.X.Y`

During the first `-rc` release, the only exception allowed for new merges is usually previously-unsupported device drivers that don't touch any other part of the kernel. It can't hurt any other part of the kernel to add them, so they are generally accepted still (it is usually the case that all new merges would just wait for the next kernel major release process).



- Requirements are determined and the patch is created
 - Preferably an open process to avoid later re-designs
- Patch is posted to the relevant mailing list
 - Peer review process
 - Subsystem maintainers and mailing list information in MAINTAINERS file
 - Most lists hosted at: <http://vger.kernel.org/vger-lists.html>
- Acceptance by subsystem maintainer
 - Not a guarantee that the patch will make the mainline
 - Generally for a final, wider audience review
- Patch is submitted to the mainline during the merge window
 - Developer must be available at this time to resolve issues that may come up
 - Only one person can do this: Linux Torvalds

Linus Torvalds is the only person who can accept patches into the mainline kernel, but this mechanism does not scale adequately (the community has seen over 10,000 patches per release!). Quite simply, its too much for Linus to do all by himself. In fact, in the 2.6.25 release, Linus personally accepted only 250 patches.

To scale better, there is a "Lieutenant" system in place, consisting of developers chosen based upon a chain of trust.

The kernel is broken down into subsystems (e.g. memory, networking, etc.) and assigned to subsystem maintainers. These maintainers are mostly responsible for accepting patches into their particular subsystem of the kernel for the mainline version. The list of subsystem maintainers and the relevant mailing list for the different subsystems are all contained in the MAINTAINERS file in the root of the source code tree.

For very large subsystems (e.g. networking), they can be broken down still further into smaller pieces (e.g. wireless networking drivers) and managed by yet another subsystem maintainer that is part of the larger chain of trust.

More aggregate subsystem maintainers can pull the patches accepted by a lower ranking subsystem maintainer into their patch list.

When the merge window opens, the subsystem maintainers ask Linus to pull all the patches they have accepted into his mainline version of the kernel.

*/usr/src/redhat/BUILD/<ver>/
maintainers*



- Problems with the development cycle that are benefited by staging trees:
 - New patches unaware of simultaneous changes from other patches that directly affect them
 - What if a developer wants to see all the patches being prepared for the next merge?
- Andrew Morton's `-mm` staging tree
 - Integrates patches from a long list of subsystem trees and those that help debugging
 - Miscellaneous patches of interest hand-selected by Andrew
 - Patches that didn't fit into a subsystem category
 - Accepted patches are forwarded to a subsystem or Linus directly
- Stephen Rothwell's `linux-next` staging tree
 - A snapshot of what the mainline is expected to look like after the next merge window closes
- Staging trees have proven to be valuable resources for finding and fixing integration problems before the beginning of the merge window

The oldest of the staging trees, Andrew Morton's `-mm` (`-mm` originally came from "memory management") is the mechanism used by about 10% of all patches accepted into the mainline kernel. The current `-mm` patch can be found at <http://kernel.org/>. The current state of the `-mm` staging tree (the so-called "`-mm` of the moment", or MMOTM tree) can be found at <http://userweb.kernel.org/~akpm/mmotm/>. Keep in mind that the `-mm` tree is bleeding-edge developmental, and there is a very real chance it won't compile!

A list of included trees in the `linux-next` staging tree can be found at <http://linux.f-seidel.de/linux-next/pmwiki/>, and they can be found at <http://www.kernel.org/pub/linux/kernel/people/sfr/linux-next/>. The `linux-next` releases are announced on the *linux-kernel* and *linux-next* mailing lists.



End of Lecture 1

- Questions and Answers
- Summary
 - Legal requirements
 - Kernel release cycle

Lab 1.1: Following the Linux Kernel Developer Community

Scenario: You want to stay up to date with Linux kernel development.

Deliverable: An understanding of what's happening in the Linux community, and a recent source tree of the "pure" Linux kernel within the `/usr/src` directory.

Instructions:

1. What is the version of the most recent stable kernel?

2.6.37

2. What are a couple of interesting topics proposed or discussed last week (okay, two weeks ago) in the Linux community? (Hint: Consult a Linux weekly news site).

lxcv.com

3. Which patches were submitted to the kernel in the last few days?

gunk mail

4. What is the most recent Fedora kernel?

2.6.28

5. What is the most interesting outstanding bug report for the Red Hat Enterprise Linux kernel?

evolution (mehes)

6. Who is the most prolific corporate contributor to the Linux kernel? (Hint: perform a Google search on "Linux contributors".)

Red hat

7. Obtain the source for a recent "pure" kernel, and expand it into the `/usr/src` directory.

2.6.37

8. If you wanted to submit a patch to the Linux `ext3` filesystem code, to whom would you email your request? What mailing lists might it be wise to monitor first?

Lab 1.1 Solutions

1. According to <http://www.kernel.org>, as of this writing, 2.6.31.
2. Try consulting <http://lwn.net>, and consult last week's kernel page.
3. Consult the vger.kernel.org kernel-list.
4. According to a mirror located from <http://mirrors.fedoraproject.org>, as of this writing, 2.6.29.6.
5. Consult <http://bugzilla.redhat.com>.
6. Modesty prevents us from providing the answer.
7. The following commands would install the Linux kernel:

```
K=http://www.kernel.org/pub/linux/kernel  
V=v2.6/linux-2.6.30.5.tar.bz2  
curl $K/$V | tar xvj -C /usr/src
```

8. According to the MAINTAINERS file, patches should be submitted to sct@redhat.com, akpm@osdl.org, and/or adilger@clusterfs.com. The `ext2-devel@lists.sourceforge.net` mailing list should be monitored before submitting any patches.

adilger - /pm
translator
package





Lecture 2

User Mode and Kernel Mode

Upon completion of this unit, you should be able to:

- Describe the role and basic architecture of the kernel
- Understand the difference between user and kernel mode
- Explain how the kernel system call API works



- A clone of the UNIX operating system, written from scratch
- Connects the application software to the hardware of a computer
- Linux is a monolithic kernel
 - Every part of the kernel runs in the same address space
- Aims for POSIX and Single UNIX Specification compliance
- Works with just about every general-purpose 32/64-bit architecture with a Paged Memory Management Unit (PMMU) and a port of the GNU C Compiler (*gcc*)

The Linux kernel is monolithic, with the advantages of simple design and no message-passing performance penalties such as those found in a microkernel architecture. The ability to dynamically load and unload modules adds some microkernel-like benefits.

In monolithic kernels, all services including process management, memory management, IPC, device drivers, filesystems, cache management, etc. are managed in kernel mode. In microkernels, only essential features like process management, memory management, and IPC run in kernel mode; all other services run as ordinary user processes that communicate with the kernel via IPC.

For more details about Single UNIX Specification compliance, see: <http://www.unix.org>.

It is frequently asked why Linux isn't rewritten in C++ or some other object-oriented language. For an extensive explanation, refer to the FAQ at <http://www.tux.org/lkml/#s153>. The quick explanation is that while not written in an object-oriented language, the Linux kernel uses object-oriented techniques in a more accessible language and form to the benefit of the massive number of contributing developers.



- Processes
 - Create, destroy, schedule
 - IPC
- Memory
 - Manage virtual address space
 - Allocate and free
- Devices
 - Manage and control
 - I/O
- Networking
 - Send/Receive/Route/Filter packets
- Filesystems
 - Abstract the different types of file systems
 - Provide structured layout

The kernel provides an environment for processes to run in. The kernel also mediates all access to hardware, and as a general design approach, attempts to represent potentially limited resources as vast and easily available.

For example, on a 32-bit system every process thinks it has 4 GiB of memory. Each process thinks it is the only process on the CPU. Processes do not need to concern themselves with any other process' traffic on a Network Interface Card.

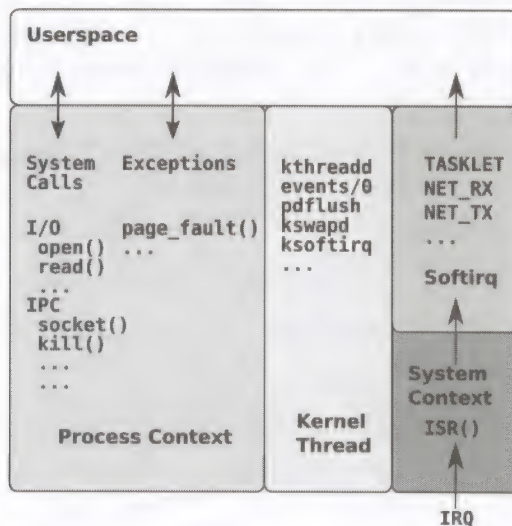
Some hardware, such as network interface cards and disk drives, are accessed through intermediate kernel layers, such as a network protocol stack or the virtual file system.



- Userspace: calculations
- Kernel: I/O, IPC, system information
- Enter the kernel via interrupts
 - System calls
 - Exceptions
 - Hardware interrupts
- Stay in the kernel via...
 - Softirqs - deferred processing while responsive to IRQs
 - Kernel threads - non-interrupt driven kernel tasks

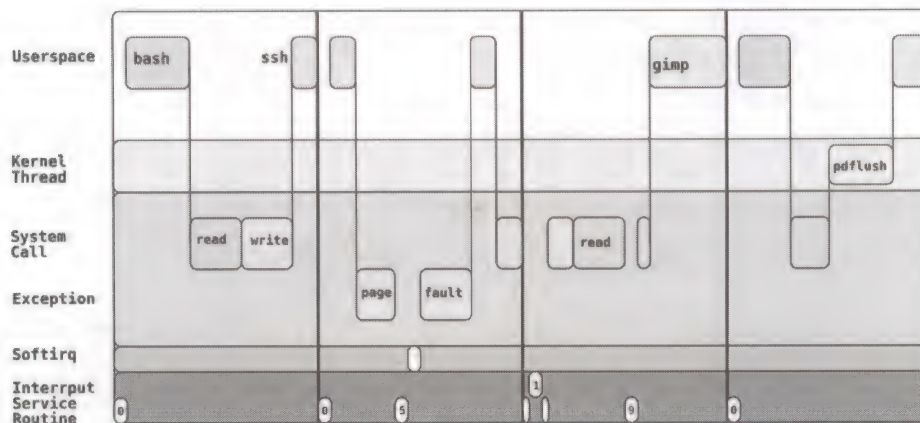
There are three ways to switch into kernel mode from user mode:

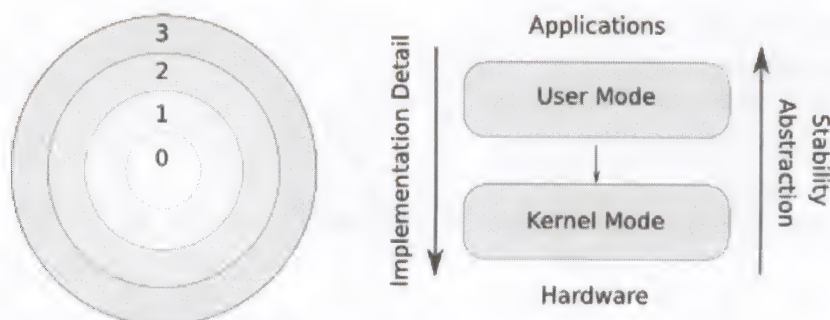
1. When a program needs the kernel to perform some service it voluntarily switches from user mode to kernel mode by executing an `int 0x80` instruction.
2. Exceptions also cause a program to switch from user mode to kernel mode. Example exceptions include a memory page fault or an illegal instruction.
3. Hardware interrupts which occur asynchronously.





- Many interwoven code paths
- Scheduling points
 - On return from system calls
 - Preemptive at clock pulse (IRQ 0)
- Kernel preemption
 - On return from ISR, kernel code path can be preempted





- Hierarchical levels or *rings* of privilege and access
 - Enforced by the CPU hardware
 - Four levels in x86 hardware
 - Ring 0 (kernel or supervisor mode) is the most privileged and has access to all devices and lower level resources
 - Ring 3 (user mode) is the least privileged mode
- The mode is a state of each CPU rather than the kernel itself

Protection rings are hierarchical levels of privilege and access. Most architectures provide enforcement of this access through hardware. On the x86 architecture, there are four ring levels. Ring 0 is the most privileged and has access to all devices. Code running in the model has a hardware mediated flag set to indicate that it is running in supervisor or kernel mode. Ring 3 (user mode) is the least privileged mode. Special gates allow a programs in a higher ring level to access a lower levels resources.

The kernel is a program that provides an abstraction layer for accessing the system's hardware devices. Only the kernel can interact with the hardware devices. It has sole access to network cards, printers, keyboards, monitors etc.

User programs read and write data from these devices by asking the kernel. Access to these devices is provided as a service by the kernel.

Memory is also a system resource. Creating or destroying a process requires kernel interaction to allocate memory for the process, and allow the process to be scheduled.



- Kernel Mode
 - Ring 0* (x86 hardware)
 - Executing code has complete and unrestricted access to the underlying hardware
 - All kernel structures and symbols are available
 - Memory is never swapped out
 - Kernel is preemptible in 2.6 and newer versions
 - On SMP systems, different processes can be running in kernel or user mode independently
 - Can execute any CPU instruction and reference any memory address
 - Kernel functions must be re-entrant
 - Used only by the lowest-level, most trusted functions of the operating system
 - Crashes in kernel mode are fatal

Recent CPU hardware from Intel and AMD offer x86 virtualization instructions for a hypervisor to control ring 0 hardware access. Intel VT (codenamed "Vanderpool", extension "vmx") and AMD-V (codenamed "Pacifica", extension "svm") CPUs utilize ring 1 so that a guest operating system can run ring 0 operations natively without affecting other guests or the host OS.

In kernel mode, the executing code has complete and unrestricted access to the underlying hardware (enforced by the CPU hardware) -- it can execute any CPU instruction and reference any memory address. Crashes in kernel mode are fatal, and will halt the operation of the entire machine. Because of this, kernel mode is reserved for the lowest-level, most trusted functions of the operating system.

For a non-preemptible kernel, the kernel cannot be preempted if the kernel is executing in process context. This is true even if the time slice for the process has elapsed. The process can only be preempted when it runs in user mode.

The kernel code can be interrupted and a change of execution flow can occur if:

- An implicit or explicit sleep occurs if some needed resource is not available
 - A device driver for a disk may block until data is ready or the resource is no longer held by another process; the process sleeps and is woken up when the resource is finally available
- An interrupt arrives when interrupt handling is enabled
 - The kernel invokes the interrupt handler and picked up where it left off afterwards

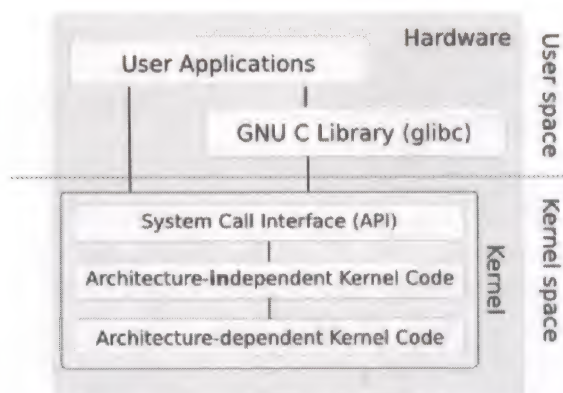
Processes always start in kernel mode, because they're started by the kernel. Whenever they are idle or waiting for I/O, they spend most of their time in kernel mode. An active process may switch back and forward between kernel mode and user mode thousands of times a second.



- User Mode
 - Ring 3 (x86 hardware)
 - Executing code cannot directly access hardware or reference memory
 - System calls are used
 - Used by most executing code
 - Crashes in user mode are always recoverable
 - Process is scheduled by, and can be preempted by, the kernel

In user mode, executing code cannot directly access hardware or reference memory (again, enforced by the CPU hardware) because it does not have privileges to do so. As a result, any access to hardware must be done through the system call API. Memory references are requested from the kernel in a similar manner. This isolation from direct access to the hardware means that crashed user mode processes are recoverable; it's just the application itself that crashed, not the entire operating system.

Because of the safety afforded by it, most processes run in user mode, most of the time.



- System calls can be thought of as special functions that manage kernel mode routines
- System calls are typically made when:
 - accessing an I/O device or file (e.g. `read()` or `write()`)
 - accessing or changing privileged data (e.g. `pid` or scheduling policy)
 - changing execution context (e.g. `fork()`'ing)
 - executing a command requiring a kernel mode function (e.g. `kill()`, `brk()`, `signal()`, etc.)

In order to do anything useful, user-space processes must be able to access services and resources from the kernel. The way this is implemented in Linux is diagrammed above. While it is possible to invoke a system call directly (using the `syscall()` function), most applications don't. Usually the user-space process makes a call to `glibc` functions which provide the system call interface that connects to the kernel and provides the transition mechanism between user and kernel space. The library functions prepare their parameters and perform all necessary work in a hardware-specific manner to interact with the kernel.

The Linux kernel can roughly be divided into three levels:

- The system call interface which acts as a bridge between user-space and the kernel
- The architecture-independent kernel code, which is common to all of the processor architectures supported by Linux
- The architecture-dependent kernel code, which is processor and platform-specific code for different architectures (sometimes more commonly referred to as a Board Support Package)



- User-space `glibc` library function actions taken
 - Stores system call parameters into registers
 - System call number is saved in `%eax`
 - Executes either `sysenter` or `int 0x80` instruction
- Kernel-space actions taken
 - `sysenter_entry()` and `system_call()` are kernel entry points
 - Entry point is determined by instruction used by `glibc`
 - Both assembly language functions manage hardware registers
 - Use `%eax` register to index `sys_call_table` array of function pointers
 - System call service routine, `sys_syscallname()`, executes

The main point of communication between the `glibc` library functions and the kernel is the `%eax` register. The C library stores a number in `%eax` corresponding to the system call to be performed. On the user-space side, the system call numbers are defined in the `asm/unistd.h` header file whose first few lines are displayed below:

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall    0
#define __NR_exit               1
#define __NR_fork               2
#define __NR_read               3
#define __NR_write              4
```

On the kernel side, the `%eax` register value is taken and used as an index into the `sys_call_table` array of function pointers in the kernel. The first few lines from `arch/i386/kernel/syscall_table.S` appear below:

```
ENTRY(sys_call_table)
    .long sys_restart_syscall    /* 0 - old "setup()" system call, ✓
used for restarting */
    .long sys_exit
    .long sys_fork
    .long sys_read
    .long sys_write
```

For obsolete or unimplemented system call numbers, their corresponding entries in `sys_call_table` point to a function called `sys_ni_syscall()` which simply returns a `-ENOSYS` value.

`sys_syscallname()` versus `do_syscallname()` functions



- Kernel-space actions taken upon return
 - System call service routine exits
 - Returns 0 or positive value on success
 - Returns a negative value on error
 - `sysenter_entry()` or `system_call()` restore registers and return
 - Executes `sysexit` or `iret` instructions respectively
- User-space `glibc` library function actions taken upon return
 - On success the system call return values are returned by the library function
 - On failure the system call return value is saved into `errno` and -1 is returned

Kernel functions that implement system calls commonly return a 0, or a positive value in some cases, when they are successful. Failure is indicated by returning a negative `errno` value. The sample code below taken from `kernel/signal.c` is the system call service routine for the `tgkill()` system call:

```
asmlinkage long sys_tgkill(int tgid, int pid, int sig)
{
    /* This is only valid for single tasks */
    if (pid <= 0 || tgid <= 0)
        return -EINVAL;

    return do_tkill(tgid, pid, sig);
}
```

*args are in the stack
not the registers*

Notice the `sys_tgkill()` system call service routing calls `do_tkill()` to actually implement `tgkill()` system call functionality. In most cases the system call service routine does basic sanity checking and handles copying of data to/from user-space data structures when needed.

The following code snippet shows what the user-space system call invocation and error handling might look like:

```
#include <errno.h>          /* definition of errno values */
...
if (tgkill(tgid, pid, SIGINT)) {
    switch (errno) {
        case EINVAL:
            ...
    }
}
```




- Upon arrival of an IRQ, the currently running task is interrupted so that the IRQ can be handled:
 1. Current task is executing normally (user mode)
 2. An IRQ arrives
 3. The currently running task is interrupted and a handler is called (transition to kernel mode)
 4. The handler code is executed (kernel mode)
 5. Upon completion of the handler, control is returned to the task (transition to user mode)
 6. The task continues to execute as if nothing happened (user mode)

Mode switches involving IRQ interrupts are very common, especially with guaranteed arrivals originating from the system clock and other hardware devices.

Interrupt handling will be discussed in more detail in a later unit.



- User processes executing system calls are much slower than a direct kernel function call
 - About 132¹ times slower using `int 80` and context switch (Pentium 4)
 - 36¹ times slower using `sysenter/sysexit` (available in 2.5.53 and newer)
- KML allows user processes to execute with privilege level of kernel mode
 - User programs can access kernel address space directly
 - System calls are very fast since mode switch is unnecessary
- Unsupported kernel patch
 - <http://web.yl.is.s.u-tokyo.ac.jp/~tosh/kml/>
 - Copy desired binaries into `/trusted` (default)
 - Y for Kernel Mode Linux when configuring kernel compile
- Recent glibc (2.3.2 and later) translates KML process system calls to kernel function calls automatically
 - System call overhead is removed without modifying the program

KML processes still have their own address space and can page and be scheduled like an ordinary user process, unlike kernel modules. Exceptions, such as segmentation faults and illegal instruction exceptions are generally handled the same as for an ordinary user mode process and don't cause kernel panics (not true for improper memory accesses or improperly executed privileged instructions).

To enable Kernel Mode Linux, specify Y for the Kernel Mode Linux field when configuring the kernel compilation. Build and install the kernel, then reboot so it is being used. Once done, all executables in the directory `/trusted` (default location) are executed in kernel mode. For example, to execute a bash shell in kernel mode, copy `/bin/bash` to directory `/trusted` (if it does not exist, create it) and execute it by specifying the fully qualified pathname to the new version of the executable.

Visit <http://web.yl.is.s.u-tokyo.ac.jp/~tosh/kml/> for the actual patches and for more information about which versions of the kernel it works on.

¹"Kernel Mode Linux: Toward an Operating System Protected by a Type Theory", Toshiyuki Maeda and Akinori Yonezawa, Lecture Notes in Computer Science, Volume 2896/2003, Pages 3-17.



End of Lecture 2

- Questions and Answers
- Summary
 - Differences between user and kernel mode
 - Execution flow switches between the two modes
 - Execution flow in process context, interrupt context and otherwise
 - System call API



Lab 2.1: Exploring Kernel Execution Contexts

Scenario: On a freshly installed machine, you will make user-space observations about various kernel execution contexts.

Instructions:

1. In a terminal window, run the command **vmstat 1**. Is the CPU spending most of its time in user mode or kernel mode?

user mode

2. What seems to be the floor for interrupts per second?

1000, thousands/s

3. Reconsider these questions while running the command **factor 895491372049813534**.

~~interrupts~~ interrupt shouldn't be a factor

4. Reconsider these questions while running the command **cat /dev/zero > /dev/null**.

again no change

5. In another terminal window, run the **top** command. From another machine, have a neighbor “ping flood” your machine with **ping -f stationX.example.com** (run as root).

While experiencing a “ping flood”, how much time does the CPU spend servicing hardware interrupts? servicing software interrupts?

1-2 % hi tsi

Lab 2.2: Exploring Kernel Transitions

Scenario: On a freshly installed machine, you will make user-space observations about user mode to kernel mode transitions.

Instructions:

1. Execute the command **time strace -c dd if=/dev/zero of=/dev/null bs=4k count=100k**. Estimate the number of system calls made per second.
~~866~~ 86265
2. Run the command **ps ax -o cmd,majflt,minflt**. Which process has had the most page faults?
sh had most min, python had most maj
3. Which process has had the most major page faults?
python -E /usr/sbin
4. Why have kernel threads incurred no page faults?
kernel space doesn't use vm.
5. Why would you expect **/sbin/init** to be a contender for the most page faults?
it's the root of the tree
6. Run the command **watch -n1 -d cat /proc/interrupts**. Other than the timer, which device is generating the most interrupts? Can you determine which IRQ line your mouse is using?
eth0, IRQ 58 mouse uses IRQ 217
7. Can you change the "niceness" of a kernel thread? The **ps axl** and **renice** commands might prove helpful.
yes
8. Run the command **dd if=/dev/zero of=/tmp/big bs=1M count=1M**. Which kernel threads become runnable during the transfer?

Lab 2.1 Solutions

1. Is the CPU spending most of its time in user mode or kernel mode?

The answer to this question depends on your current system activity. The relevant statistics displayed by **vmstat** are found in the `cpu/us` and the `cpu/sy` columns. The first line of output produced by **vmstat** is an overall total/average calculated since system boot, the remaining lines of output reflect current system activity.

2. What seems to be the floor for interrupts per second?

A standard kernel will not drop below 1000 interrupts per second. A xen kernel will not drop below 250 interrupts per second.

3. While running **factor**, most CPU consumption will be made in user mode since the command is performing many numerical calculations. Interrupts should not show any noticeable change.
4. The **cat** command copies data between two memory-based devices. CPU consumption should increase in both user and system context, but system mode should get the majority of CPU time. Again, interrupts should not show any noticeable change since `/dev/zero` and `/dev/null` are memory-based.
5. While experiencing a “ping flood”, how much time does the CPU spend servicing hardware interrupts? servicing software interrupts?

These measurements are displayed in the `%hi` and `%si` fields of **top**. On the course author's test system, hardware interrupts (`%hi`) took about 6% of the CPU and software interrupts (`%si`) consumed about 8%.

Lab 2.2 Solutions

1. Execute the command **time strace -c dd if=/dev/zero of=/dev/null bs=4k count=100k** and estimate the number of system calls made per second.

To calculate the average number of system calls per second, take the total number of system calls returned by the **strace -c** command and divide that value by the `real` value printed by the **time** command.

2. If your system is running a graphical desktop, then the X server (**Xorg**) is the most likely candidate for the most page faults.
3. Which process has had the most major page faults?

The answer to this question depends on your system activity. It is likely that **init** ranks high in the list of processes.

4. Kernel threads don't incur page faults because they don't use their own virtual memory.
5. **/sbin/init** is a top contender for page faults because it is idle most of the time which makes it a prime candidate for being swapped out of memory.
6. Other than the timer, the device which generates the most interrupts is most likely a storage device or a network card. Again, the answer to this question depends upon your system's current activity.

Your mouse is generating interrupts on the IRQ either one of your USB devices or on a device associated with an `i8042` (which is the PS/2 port).

7. You *can* use **renice** to change the “niceness” of a kernel thread.
8. The **dd if=/dev/zero of=/tmp/big bs=1M count=1M** command generates considerable disk activity. The kernel threads which become runnable during the transfer are primarily **kjournald**, secondarily **kswapd**, and occasionally **kblockd**.





Kernel Compilation and Tools

Upon completion of this unit, you should be able to:

- Compile a new version of the kernel
- Install a new kernel and boot from it
- Create and customize an initial RAM disk image
- Efficiently search kernel source code using common tools



- **kernel package variants:**
 - `kernel-xen`
 - `kernel-PAE`
 - `kernel-debug`
 - `kernel{,-xen,-PAE}-debuginfo`
 - `kernel-debuginfo-common`
- **kernel*-devel**
 - Provides kernel headers and makefiles sufficient to build modules against the same-versioned kernel package
 - `/usr/src/kernels/<version>`
- **kernel-headers**
 - Header files specifying interface between kernel and userspace libraries and programs

There are several variants of the `kernel` package. The `kernel-xen` package is precompiled with the ability to run in a Xen virtual machine. The `kernel-PAE` package has a 32-bit version of the kernel that can address greater than 4GB of memory and supports up to 64GB of memory using Physical Address Extensions.

The `kernel-debug` package is compiled with numerous debugging options enabled, and should only be used when debugging the kernel as some of the options impact performance noticeably.

The `kernel-debuginfo` package (and its PAE and Xen variants) was compiled with the `CONFIG_DEBUG_INFO` build option, producing kernel and module code containing symbol information for debugging purposes. As a result, these are relatively large (about 150MB) compared with other kernel packages. This package is required by SystemTap. To install these packages through RHN, the `rhel-debuginfo` repo needs to be enabled either in the `/etc/yum.repos.d/rhel-debuginfo.repo` file (`enabled=1`) or on the command line at install time (e.g. **`yum --enablerepo=rhel-debuginfo -y install kernel-debuginfo`**). These packages are also available via anonymous ftp from `redhat.com` (e.g. `ftp://redhat.com/pub/redhat/linux/enterprise/5Server/en/os/i386/Debuginfo`).

The `kernel-debuginfo-common` package, also provided by the `rhel-debuginfo` repo, provides the kernel source files common to all builds and is required by the `kernel-debuginfo` package.

The `kernel-devel` package provides kernel headers and makefiles sufficient to build modules against the kernel package.

The `kernel-headers` package includes the C header files that specify the interface between the kernel and userspace libraries and programs, and define structures and constants needed for building most programs.

The `kernel-doc` package installs just the `Documentation` subdirectory of the kernel source code tree into `/usr/share/doc/kernel-<version>` directory.



- `kernel-A.B.C-EXTRAVERSION.arch.rpm`
 - `A.B` = `VERSION.PATCHLEVEL` (major release number of kernel)
 - `C` = `SUBLEVEL` (minor release number of kernel)
 - The `EXTRAVERSION` specifies customizations of `A.B.C` version
- The kernel version is determined by Makefile definitions

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 18
EXTRAVERSION = -47.el5
```

- The Makefile has a rule which creates the `linux/version.h` header file

The *A* version number is rarely updated (in fact, only twice in the kernel's lifetime (1.0 in 1994 and 2.0 in 1996)). The *B* version number denotes the major revision version of the kernel, and the *C* number denotes the minor revision of the kernel (security patches, bug fixes, new features, new drivers, etc.).

Red Hat kernels have a base release version (the original kernel version that was started with) and a Red Hat release version number (indicating the patch level). The base release version consists of 3 digits: the first two indicate the major release and the third digit indicates the minor release number. Prior to kernel version 2.6.8, even-numbered major versions indicated a stable release of the kernel, but that is no longer the case.

During the kernel build process the top-level kernel Makefile creates the `linux/version.h` include file. It defines a few useful macros that are used to compare kernel versions during the build process. The following is the contents of `linux/version.h` on a Red Hat Enterprise Linux 5.4 system (note the additional custom Red Hat macros that are available):

```
#define LINUX_VERSION_CODE 132626
#define KERNEL_VERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c))
#define RHEL_MAJOR 5
#define RHEL_MINOR 4
#define RHEL_RELEASE_CODE 1284
#define RHEL_RELEASE_VERSION(a,b) (((a) << 8) + (b))
```

The `KERNEL_VERSION()` macro can be used to generate a single numeric value that represents the kernel version for comparison with the current kernel, `LINUX_VERSION_CODE`.

The current version of the kernel can be obtained by executing:

```
$ uname -r
```



- The source code is always the most current and best documentation
- `kernel-doc` package
 - Documentation files from kernel source
 - `/usr/share/doc/kernel-doc-version/Documentation`
- Docbook documentation:
 - Formats: xml, sgml, ps, pdf, html, man

```
cd /usr/src/redhat/BUILD/kernel-version/linux-version.arch  
make pdfdocs
```

The kernel source code is always the best and most up-to-date place to find information about the kernel.

The `kernel-doc` package provides the `Documentation` subdirectory from the kernel source and installs it below the `/usr/share/doc/kernel-doc-version` directory. This subdirectory contains many helpful files that cover a broad range of topics concerning the kernel and is useful even if the rest of the kernel source isn't installed or needed.

There also exists Docbook versions of the kernel documentation that can be created in XML, SGML, PostScript, PDF, HTML, and manual page formats. The books that will be created from XML are located in the `Documentation/DocBook` subdirectory.

For example, the following commands first change to the proper source code directory, then build HTML-based kernel documentation:

```
# cd /usr/src/redhat/BUILD/kernel-version/linux-version.arch  
# make htmldocs
```




- `Makefile`
- `Documentation/` - text-based kernel information files
- `arch/` - all arch-specific code
- `block/` - block device layer support
- `configs/` - arch-specific `.config` files
- `crypto/` - cryptographic API for use by kernel itself
- `drivers/` - driver code for most peripheral devices
- `fs/` - VFS and filesystem-specific code
- `include/` - most header files
- `init/` - provides "early userspace" environment

Helpful information about the contents of most subdirectories is available by reading the `Kconfig` file in each directory (that has one). Sometimes the files in each directory are named in such a way as to be obvious about their contents. The `Kconfig` file is used by kernel configuration tools to display configuration options and provide help text about each configuration option.

The `Makefile` in the top-level directory of the source tree sets many rules and variables, and is used for compiling the kernel and its modules.

All architecture-specific coding is contained in the `arch` directory and the `include/asm-arch` directory.

The `block` directory provides the block layer coding and permits the block layer to be removed from the compiled kernel (for example, some embedded devices).

Most header files found at the beginning of C kernel code are found in the `include` directory.

`init` provides the "early userspace" environment: that which needs to be available while a kernel is loading but not in the kernel itself.



- `ipc/` - code for different types of IPC
- `kernel/` - generic kernel code that doesn't fit anywhere else
- `lib/` - routines of generic use to all kernel code
- `mm/` - high-level memory management code
- `net/` - high-level networking code
- `samples/` - a place to build and test sample kernel code
- `scripts/` - scripts useful for building the kernel
- `security/` - Linux security models code (e.g. SELinux)
- `sound/` - sound card support code
- `usr/` - `initramfs` support for "early userspace" root filesystem

The `ipc` directory contains code for Inter-Process Communication routines, such as shared memory, semaphores, etc.

The `kernel` directory holds kernel code that doesn't fit neatly anywhere else in the source tree. Examples include the scheduler, `printk()`, signal handling, etc.

The `mm` directory holds high-level memory management code to implement virtual memory along with lower-level code that is architecture specific in `arch/arch/mm/`.

The `net` directory contains high-level networking code that routes packets to and from low-level network card device driver code. Depending upon the data, it either passes the packets on to user-level applications, the kernel itself, or discards the data.



- The steps involved in preparing a new Red Hat kernel are generally as follows:
 1. Install kernel development tool packages
 2. Install kernel source package
 3. Prepare source code for compile (extract tarball and apply patches)
 4. Edit `Makefile` for `EXTRAVERSION`
 5. Set compilation options in `.config`
 6. Compile the kernel and modules
 7. Install the following:
 - modules
 - compressed kernel
 - system map
 - kernel compilation options config file
 8. Build a new `initrd` image
 9. Update `grub.conf`

We will discuss each step in more detail on the following slides.



- Bare minimum package install (including dependencies):

```
yum -y install rpm-build gcc redhat-rpm-config ✓  
ncurses-devel
```

- For a more complete development environment:

```
yum -y groupinstall "Development Tools"  
yum -y install ncurses-devel
```

To list packages included in the Development Tools group (also known as `development-tools`), use the following command:

```
# yum groupinfo "Development Tools"
```

To list already-installed and available package groups, use the following command:

```
# yum grouplist
```

The group names output by the previous command can be used in the **yum** command, but double quotes are often needed to protect the space(s) in the group name.

The additional package groups "GNOME Software Development" (`gnome-software-development`) and "KDE Software Development" (`kde-software-development`) contain the packages needed for GUI-based kernel compilation configuration tools, such as those presented when running the **make gconfig** and **make xconfig** commands, respectively.

The `ncurses` and `ncurses-devel` packages are needed for the **make menuconfig** command, but the `ncurses-devel` package is not installed by the **yum -y groupinstall "Development Tools"** command so make sure to do it manually.



- Obtaining Red Hat kernel source:
 - Red Hat Network
 - **yumdownloader --source kernel**
 - The **yumdownloader** utility is included in the **yum-utils** package
- Installing kernel source
 - **rpm -ivh kernel-version.src.rpm**
 - Installs into `/usr/src/redhat/{SOURCES,SPECS}`
 - Requires **rpm-build** package

When a Red Hat Enterprise Linux kernel source RPM is installed, it places the original source code and all patches in `/usr/src/redhat/SOURCES`. It also places an RPM spec file in `/usr/src/redhat/SPECS` that can be used to build the code after applying all the patches.

We will not cover the "vanilla" kernel in this course, but stable, developmental, and experimental (-mm series by Andrew Morton) versions of it can be downloaded from the site <http://www.kernel.org>. The latest snapshot of the different "vanilla" kernel versions can be viewed at:

http://www.kernel.org/kdist/finger_banner

or by using the command:

```
$ finger @www.kernel.org
The latest stable version of the Linux kernel is:      2.6.28.7
The latest prepatch for the stable Linux kernel tree is: 2.6.29-rc8
The latest snapshot for the stable Linux kernel tree is: 2.6.29-rc8-git2
The latest 2.4 version of the Linux kernel is:         2.4.37
The latest 2.2 version of the Linux kernel is:         2.2.26
The latest prepatch for the 2.2 Linux kernel tree is:  2.2.27-rc2
The latest -mm patch to the stable Linux kernels is:   2.6.28-rc2-mm1
```



- Extract source code and apply patches

```
# cd /usr/src/redhat/SPECS
# rpmbuild -bp --target=$(uname -m) kernel-version.spec
```

- Build directory:

```
/usr/src/redhat/BUILD/kernel-version/linux-version.arch
```

The **rpmbuild** command is used to unpack the source code to a build directory and apply patches (-bp) according to its spec file, then prepare the default compilation options for the current target architecture (--target=\$(uname -m)).

The pre-patched "vanilla" source code is in /usr/src/redhat/BUILD/kernel-version/vanilla, while the patched version of the source code is created in /usr/src/redhat/BUILD/kernel-version/linux-version.arch.

To manually apply a patch to a kernel depends a lot on how the patch was created, and specifically from which directory the patch was generated. In general, patches are created in /usr/src/redhat/BUILD/kernel-version.arch and are **diff'd** against the current source tree or a pristine original (/usr/src/redhat/BUILD/kernel-version/vanilla) using a command such as:

```
# diff -uNr kernel-2.6.18-original kernel-2.6.18-edited > patchfile
```

and applied to a source tree by changing into the top level of the source tree to be patched and using a command such as:

```
# patch -p1 < patchfile
```

or, if compressed:

```
# bzip2 -dc patchfile.bz2 | patch -p1
```

The --dry-run option to **patch** is useful for testing the patch file before actually applying it, because backing out a partially applied patchfile can get messy. If a patch is applied successfully, passing the -R option and original patch file to the **patch** command is useful for reversing the patch.



- The Makefile in the build directory contains the following lines:

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 18
EXTRAVERSION = -prep
RHEL_MAJOR = 5
RHEL_MINOR = 1
NAME=Avast! A bilge rat!
```

- EXTRAVERSION is often customized to indicate the particular nature of the custom kernel
 - Example: `kernel-2.6.18-1.0.custom`
 - See also the configuration tool menu's type-in widget:

General Setup -> Local version - append to kernel release

The VERSION, PATCHLEVEL, SUBLEVEL, and EXTRAVERSION numbers, as a whole, are used wherever a version number is automatically inserted into an object's name by some element of the kernel compilation or installation process. For example, the subdirectory in `/lib/modules` holding all of that kernel's compiled modules is built from those numbers and characters.

These variables are also used in a define statement in `include/linux/version.h` using the algorithms:

```
#define LINUX_VERSION_CODE $(shell expr $(VERSION) \* 65536 + \
$(PATCHLEVEL) \* 256 + $(SUBLEVEL));
#define KERNEL_VERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c))
```

where a, b and c refer to the >VERSION, PATCHLEVEL, and SUBLEVEL, respectively. These defines can be queried within kernel and module source code for the current version of the kernel.

Similarly, the RHEL_MAJOR and RHEL_MINOR variables are also defined in `include/linux/version.h` and can be used within the kernel or a module to query for the specific release of Red Hat operating system:

```
#define RHEL_RELEASE_CODE $(shell expr $(RHEL_MAJOR) \* 256 + \
$(RHEL_MINOR));
#define RHEL_RELEASE_VERSION(a,b) (((a) << 8) + (b))
```

where a and b are RHEL_MAJOR and RHEL_MINOR, respectively.

In case you are wondering where the NAME quote comes from...the 2.6.18 version of the kernel was released by Linus Torvalds on "Talk like a Pirate Day" (<http://talklikeapirate.com/>) with the exclamation "*She's good to go, hoist anchor!*". The NAME variable isn't actually used anywhere.



- Thousands of compilation options
- Option configuration tools:
 - **make oldconfig**
 - Only asks the configuration questions which are not already answered in `.config`
 - **make menuconfig** (text-based menu)
 - **make gconfig** (GNOME X window)
 - **make xconfig** (KDE X window)
- The configuration tools produce `.config` in the top-level directory of kernel source code
 - Contains all set and unset options
 - `CONFIG_X=m` means that `X` is dynamically loadable

The thousands of compiler configuration options are most easily configured using one of the **make** tools above, which ultimately produce a file named `.config` in the top-level of the kernel source tree containing all configured options.

Not listed above is **make config**, a very limited configuration tool. When invoked, it serially queries the developer for each and every kernel compiler option, one at a time. This is a very impractical method of configuring a kernel with potentially thousands of compile-time options and should only be used as a last resort.

make oldconfig only queries the developer for compiler options which are not already answered in the `.config` file. This is useful for upgrading to a more recent version of the kernel, for example.

make menuconfig, the tool most often used, presents the kernel compilation options using a text-based menu-style interface that is based on the `ncurses` library. It will read in existing settings in `.config` and allow changes to be made.

make gconfig and **make xconfig** use GNOME-based and KDE-based graphical window tools, and therefore require the X Window system, to configure each kernel compilation option.

The `ncurses` and `ncurses-devel` packages are needed for the **make menuconfig** command, but the `ncurses-devel` package is not installed by the **yum -y groupinstall "Development Tools"** command so make sure to do it manually.



- The most time consuming step of creating a new kernel
- **make** is equivalent to:
 - **make bzImage**
 - **make modules**

All **make** commands are run in the `/usr/src/redhat/BUILD/kernel-version/linux-version.arch` directory, which is the top of the kernel source tree.

The **make bzImage** command compiles the kernel source code, and the **make modules** command is used to compile the kernel modules.



- **make modules_install**

- Creates subdirectory under `/lib/modules` named *version*
- Moves the kernel-specific modules to the new subdirectory
- Executes: **depmod -ae -F System.map version**
 - Creates/Updates `/lib/modules/version/modules.dep`

In the above slide, *version* = `${VERSION}.${PATCHLEVEL}.${SUBLEVEL}${EXTRAVERSION}`.

The **depmod** command above is used to create a file named `/lib/modules/version/modules.dep` containing a list of module dependencies. Module dependencies are determined by reading each module in the kernel-specific directory structure and determining which symbols it exports and which it needs. By referencing the `System.map` produced when the kernel was built, unresolved symbol references can be identified and reported.



- **make install** is equivalent to:

```
# cp arch/platform/boot/bzImage /boot/vmlinuz-version
# cp System.map /boot/System.map-version
# mkinitrd -v -f /boot/initrd-version.img version
# grubby --copy-default --add-kernel=/boot/vmlinuz-version ✓
--initrd=/boot/initrd-version.img --title="Red Hat ✓
Enterprise Linux Server (version)"
```

- Optionally, copy `.config` to `/boot` (provides a record of compiler options used)

```
# cp .config /boot/config-version
```

In the above commands, `version = ${VERSION}.${PATCHLEVEL}.${SUBLEVEL}${EXTRAVERSION}`.

Executing **make install** in the root directory of the source code is a shortcut to several commands that otherwise would be needed to implement the newly compiled kernel. The compressed version of the newly compiled kernel and its corresponding system map must be copied into the `/boot` directory, a new initial RAM disk is created with the required modules, and GRUB's configuration file must be updated with a new entry describing how to boot the new kernel.

The `System.map` file is produced by running `nm` on the Linux kernel binary, `vmlinux`, and is an on-disk representation of `/proc/kallsyms`, containing all symbol names, types, and addresses of `vmlinux`. Its primary use is in debugging. If a kernel "oops" message appears, the utility `ksymoops` can be used to decode the message into something useful for developers. `ksymoops` makes use of the `System.map` to map PC values to symbolic values. Note that 2.5 kernels have an in-kernel oops decoder called `kksymoops`, which does not need `System.map`. Utilities (like `ps -l`) use `System.map` to determine the `WCHAN` field and look for it in a set of standard places, such as `/boot/System.map-version`.

The **mkinitrd** command creates a new initial RAM disk containing, among other things, kernel-specific modules needed at boot time (e.g. filesystem and block device drivers needed to load the root filesystem) that aren't already statically compiled into the kernel. The initial RAM disk file created by **mkinitrd** is a gzip-compressed `cpio` archive. To view its contents:

```
# gzip -dc initrd-version.img | cpio -t
```

The contents of the image can be manually edited by first extracting it to a directory, making the changes, re-archiving the image, and finally moving and renaming it appropriately for its `grub.conf` configuration:

```
# mkdir build; cd build
# gzip -dc ../initrd-version.img | cpio -ivd
(make edits...)
# find . | cpio -co | gzip -c9 > /boot/initrd-version-test.img
```

Additional modules can also be forcibly added to the image using the **mkinitrd** command itself. This is useful if, for example, you intend to make changes that you have not yet implemented. Two methods that can be used to force a module into the initial RAM disk using **mkinitrd**:

1. Using the `--with` option, for example:

```
mkinitrd --with=scsi_mod /boot/initrd-version.img version
```

Neither the **.ko** suffix nor the full path name to the module are necessary.

2. Placing a filesystem-related directive in `/etc/modprobe.conf`, for example:

```
echo "alias scsi_hostadapter qla2300" >> /etc/modprobe.conf
mkinitrd /boot/initrd-version.img version
```

The commands above will result in at least the `qla2300` module (plus any dependent modules) to be loaded into the initial RAM disk when **mkinitrd** is run.

Note that the **grubby** command will copy kernel arguments from the current kernel (`--copy-default`) and add a new entry to `/boot/grub/grub.conf` for the newly compiled kernel version, but will not make it the default entry.

While optional, it can be useful to have a record of what options were used in the compilation of a kernel version. Copying the `.config` file used to configure a kernel to the `/boot` directory with an appropriate name is a common practice for preserving this information.



- The set of in-kernel symbols used by drivers and other kernel modules
- Does not refer to the user-space program syscall interface
- There are frequent changes to the upstream kernel kABI
 - `Documentation/stable_api_nonsense.txt`
- Red Hat avoids any changes to the kABI
 - Otherwise risk a 3rd-party module failing to load after a kernel update
 - Customized modules are most affected
- RPM signatures help minimize the problem

http://www.kroah.com/log/linux/stable_api_nonsense.html



- Tool for browsing source code efficiently and quickly
- Generates search results index for faster future searches
 - **cscope -Rk** (or **make cscope**)
- Text-based interface using `ncurses` library
- Searches source code for:
 - All references to a symbol
 - Global definitions
 - Functions called by or calling a function
 - Text strings (can do "fuzzy" parsing)
 - File names and files that include a file
 - Regular expressions

The first time `cscope` is run, an index file containing the search results must be created. From the root of the source code tree, execute:

```
# cscope -Rk
```

```
Cscope version 15.6
```

```
Press the ? key for help
```

```
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Find all function definitions:
Find all symbol assignments:
```

For the specific case of building a `cscope` index database for a kernel source tree, one could also have executed the command `make cscope` from the source's root directory. Subsequent invocations will rebuild the cross-reference again only if a source file or the list of files has changed.

For more information about `cscope`:

```
http://cscope.sourceforge.net/
```

```
http://cscope.sourceforge.net/cscope\_man\_page.html
```

```
http://cscope.sourceforge.net/cscope\_vim\_tutorial.html
```

Handwritten notes:
for 1 + 1 > - if 1/2
90% - you are - already
cscope -d - with



- Formerly known as the "Linux Cross Referencer"
- Web-based kernel source code browser
- Provides links to the definition and usage of any identifier
- Several packages required for installation, including:
 - `httpd`, `perl`, `postgresql`, and `git`
- Online version for Red Hat kernels: <http://rhkernel.org>

LXR (which was formerly known as the "Linux Cross Referencer"), is a web-based tool which indexes and presents a source code repository. While originally designed for browsing Linux kernel source code, other software projects have found it useful. LXR's main author forked a new version, named LXRng.

Information about LXR can be viewed at <http://lxr.linux.no/>, and the latest version of LXR can be downloaded using **git**:

```
# git clone git://lxr.linux.no/git/lxrng.git
```

Several packages are required by LXR including `httpd`, `perl`, `git`, and `postgresql`.



- Revision control system
- Designed by Linus Torvalds
 - Specifically for managing Linux kernel development
- Can import from other revision control systems, including:
 - CVS
 - Subversion

The git revision control utility is tuned for kernel development. It performs quite well when dealing with large repositories and large numbers of patches.

Git alternatives/complements:

- Mercurial (<http://www.selenic.com/mercurial/>) - among non-git using kernel developers, Mercurial, which shares many features with git, but provides an easy to use interface, is the next most popular choice.
- Quilt (<http://savannah.nongnu.org/projects/quilt/>) - not a source code management system, but rather a patch management system. It does not track change histories over time, but rather tracks a specific set of changes against an evolving code base and works very well with staging trees like Andrew Morton's -mm.

Once git is installed, it is useful to configure git to know your name and e-mail address for log information. This configuration can be done on the command-line with git itself:

```
[user@host]$ git config --global user.name 'Joe User'
[user@host]$ git config --global user.email 'user@host.domain.com'
```

Another option is to create a file called `~/.gitconfig` with the following lines:

```
[user]
    name 'Joe User'
    email 'user@host.domain.com'
```




- Local git documentation:
 - `gittutorial(7)` man-page
 - `/usr/share/doc/git-version`
- Documentation available on the web:
 - **git** website:
 - <http://git-scm.com/>
 - *Kernel Hacker's Guide to git*:
 - <http://linux.yyz.us/git-howto.html>

Check out the `gittutorial(7)` man-page to get started using **git**. More extensive reference documentation can be found in the `git-version` subdirectory under `/usr/share/doc`.

Other excellent documentation is available on the **git** website: <http://git-scm.com/documentation> or <http://www.kernel.org/pub/software/scm/git/docs/user-manual.html>.

Jeff Garzik maintains the *Kernel Hacker's Guide to git*, which has information specific to kernel development: <http://linux.yyz.us/git-howto.html>.



End of Lecture 3

- Questions and Answers
- Summary
 - Various kernel packages
 - Kernel building and installation
 - Useful kernel programming tools
 - `cscope`
 - LXR
 - `git`



Lab 3.1: Compiling the Red Hat Kernel

Scenario: We will re-compile and implement the Red Hat kernel.

Deliverable: The system is running on a custom kernel.

Instructions:

1. Update your kernel packages, make sure the updated version will be the default when you reboot, then reboot. Once the machine is back up, make sure you are running the updated version of the kernel. *- 2.6.18-164.el5 PAE ✓*
2. Install the kernel development environment RPMs you need (note: they may already be installed). *- nothing to do*
3. Download and install the current kernel's source RPM. *yum install yum-utils*
4. Extract the kernel source RPM to the `/usr/src/redhat/BUILD` directory and apply all patches. ✓
5. Change directory to the top-most directory of the new kernel source tree, and change the `EXTRAVERSION` parameter in the Makefile to `"-test1"`. ✓
6. Explore the kernel's configuration using the **"make menuconfig"** tool. Enable "Kernel .config support" in the kernel. What does this do, and how could you find out? ✓
7. Compile the new kernel.
8. When the compilation completes (10-15min depending upon system speed), install the newly compiled kernel modules.
9. Install the new custom kernel so that it is the default at boot time, copy the edited configuration into place for future reference, and reboot.
10. Verify the new kernel works and that the configuration is available at `/proc/config.gz`.

Lab 3.2: Searching Kernel Source with cscope

Deliverable: Working knowledge of the **cscope** tool.

Instructions:

1. Make sure **cscope** is installed and use it to index your Linux source tree.

2. What is `EEPROM_BAD_CSUM_ALLOW` and where is it defined?

drivers/net/ethernet/ibm/e100.c "Allow bad eeprom checksums"

3. Which files contain references to the `do_fork()` function?

process.c, process-xen.c, process-kernel.c, sys_admin.c

4. Which file contains the text string: `config.gz`?

configs.c

5. Spend some more time becoming familiar with **cscope**. When you are finished press **Ctrl-d** to exit **cscope**.

Optional Lab 3.3: Introducing LXR

Deliverable: A basic familiarity with the capabilities provided by LXR

Instructions:

1. If Internet access is provided to the classroom, use a web browser and access the LXR website. The URI you use to access LXR is `http://lxr.linux.no/`.
2. Navigate the website and find the pristine source code for the kernel you are using. Spend about 10 or 15 minutes browsing the source code and discover some of the capabilities of LXR. Note how useful the hyperlinks are in the source code itself.

Challenge Lab 3.4: Managing Revisions with git

Deliverable: A working **git** repository

Instructions:

1. Install **git** on your machine. This should pull in some dependencies as well.
2. Setup **git** to automatically tag your commits with your own name and email-address.
3. Create a new directory `~/Projects/gitexample` and initialize it as a **git** repository.
4. Now try adding some files to your new repository and tracking some changes to these files. You can create files by hand or copy in something like `/etc/sysconfig/*`.
5. Read through the `gittutorial(7)` man-page and experiment with making and committing changes to some of the files in your repository. View the differences between previous versions and log entries explaining your revisions. Create and manage branches if time permits.

Lab 3.1 Solutions

1. Update your kernel packages, make sure the updated version will be the default when you reboot, then reboot.

```
[root@host ~]# yum -y update kernel\*
[root@host ~]# cat /etc/grub.conf
```

Make sure that the default parameter points to the updated kernel (0 selects first kernel stanza, 1 selects second kernel stanza).

```
[root@host ~]# reboot
```

Once the machine is back up, make sure you are running the updated version of the kernel.

```
[root@host ~]# uname -r
```

2. Install the kernel development environment RPMs you need (note: they may already be installed).

```
[root@host ~]# yum -y groupinstall "Development
Tools"
[root@host ~]# yum -y install ncurses-devel
```

(The ncurses-devel package is needed for the **make menuconfig** command.)

3. Download and install the current kernel's source RPM.

```
[root@host ~]# yum -y install yum-utils
[root@host ~]# yumdownloader --source kernel
[root@host ~]# rpm -ivh kernel*.src.rpm
```

4. Extract the kernel source RPM to the /usr/src/redhat/BUILD directory and apply all patches.

```
[root@host ~]# cd /usr/src/redhat/SPECS
[root@host SPECS]# rpmbuild -bp kernel-version.spe
c
```

5. Change directory to the top-most directory of the new kernel source tree, and change the EXTRAVERSION parameter in the Makefile to "-test1".

```
[root@host SPECS]# cd ../BUILD/kernel-version/linu
x-version
[root@host linux-version]# vim Makefile
```

In the Makefile, change the line that reads:

```
EXTRAVERSION = -prep
```

to instead read:


```
EXTRAVERSION = -test
```

6. Explore the kernel's configuration using the "**make menuconfig**" tool. Enable "Kernel .config support" in the kernel. What does this do, and how could you find out?

```
[root@host linux-version]# make menuconfig
```

In the configuration tool, scroll to "General Setup" and press the Enter key to see the submenu. In the submenu, scroll to "Kernel .config support" and press the 'h' key to see a help menu that describes what this option does. Exit the help menu and press the 'y' key (an asterisk should appear in the square brackets). Also enable "Enable access to .config through /proc/config.gz" just below it. Exit the configuration tool and save the new configuration.

7. Compile the new kernel.

```
[root@host linux-version]# make
```

8. When the compilation completes (10-15min depending upon system speed), install the newly compiled kernel modules.

```
[root@host linux-version]# make modules_install
```

9. Install the new custom kernel so that it is the default at boot time, copy the edited configuration into place for future reference, and reboot.

```
[root@host linux-version]# make install
```

```
[root@host linux-version]# cp .config ✓  
/boot/config-version
```

```
[root@host linux-version]# reboot
```

10. Verify the new kernel works and that the configuration is available at /proc/config.gz.

```
[root@host ~]# uname -r
```

```
[root@host ~]# less /proc/config.gz
```

Lab 3.2 Solutions

1. Make sure **cscope** is installed and use it to index your Linux source tree.

```
[root@host ~]# yum list cscope
[root@host ~]# cd /usr/src/redhat/BUILD/kernel-version/linux-version
[root@host linux-version]# cscope -Rk
```

2. What is `eeeprom_bad_csum_allow` and where is it defined?

Navigate to "Find this global definition:", type "eeeprom_bad_csum_allow", then press the *Enter* key. The `eeeprom_bad_csum_allow` variable is defined in `drivers/net/e100.c`.

3. Which files contain references to the `do_fork()` function?

Navigate to "Find this C symbol:", then enter "do_fork". This functions should be found in `sched.h`, `process-xen.c`, `process.c`, and `fork.c`.

4. Which file contains the text string: `config.gz`?

Navigate to "Find this text string:", then enter "config.gz". The string should have been found in the file `configs.c`.

5. Spend some more time becoming familiar with **cscope**. When you are finished press *Ctrl-d* to exit **cscope**.

Challenge Lab 3.4 Solutions

1. Installing **git** on your machine can be done by running the command: **yum install git**.
2. Setup **git** to automatically tag your commits with your own name and email-address.

Edit the file `~/.gitconfig` to look something like:

```
[user]
    name = Malcolm Reynolds
    email = captain@firefly.ship
```

3. Create a new directory `~/Projects/gitexample` and initialize it as a **git** repository:

```
[user@host ~]$ mkdir -p ~/Projects/gitexample
[user@host ~]$ cd ~/Projects/gitexample
[user@host gitexample]$ git init
Initialized empty Git repository in 
/home/user/Projects/gitexample/.git/
```

4. Now try adding some files to your new repository and tracking some changes to these files. You can create files by hand or copy in something like `/etc/sysconfig/*`:

```
[user@host gitexample]$ cp -a /etc/sysconfig .
... Warnings about unreadable files omitted ...
[user@host gitexample]$ git add .
[user@host gitexample]$ git commit
```

After entering a log message using a text editor you should see output similar to the following:

```
[master (root-commit) 2e0ceb0] Initial entry into 
git
108 files changed, 6699 insertions(+), 0 
deletions(-)
create mode 100644 sysconfig/atd
create mode 100644 sysconfig/authconfig
... Remaining file listings omitted ...
```

5. Read through the `gittutorial(7)` man-page and experiment with making and committing changes to some of the files in your repository. View the differences between previous versions and log entries explaining your revisions. Create and manage branches if time permits.



Lecture 4

Modules

Upon completion of this unit, you should be able to:

- Understand how kernel modules are created and interact with the kernel



- Modules are small kernel extensions that may be loaded and unloaded at will
- Always operate in kernel space
- Can implement drivers, filesystems, firewall, and more
- Are located under `/lib/modules/${uname -r}/`
- Compiled for a specific kernel version and are provided with the kernel RPM
- Third party modules may be added

A kernel module is an optional portion of kernel code that may be loaded after the kernel has been initialized. Only a few modules that are essential to all systems are compiled directly into the kernel. Dynamic modules that are needed at boot time are loaded by grub from the initrd (initial RAM disk). Other modules may be loaded as needed later and are found in the `/lib/modules/` directory.

Modules are used for several reasons:

- Reduced memory footprint: memory is not used for drivers that are not required.
- Flexibility: modules may be added to the system after it has been installed. These modules are often called third-party drivers.
- Maximizing uptime: a module may be unloaded and reloaded as many times as wanted with no reboot.



- All are part of `module-init-tools` package
- **lsmod** provides a list of loaded modules from `/proc/modules`
- **insmod** loads a module
 - Requires path
- **rmmod** unloads a module
- **modprobe** can load and unload modules, including dependencies
 - `-r` option to unload
 - Uses **depmod**-generated file for dependency information
- **depmod** creates a list of module dependencies
 - `/lib/modules/version/modules.dep`
 - Also generates hotplug infrastructure map files
 - Example: `/lib/modules/version/modules.usbmap`
- **modinfo** displays information about any available module
- `/etc/modprobe.conf` and `/etc/modprobe.d/` are used for module configuration:
 - Parameters to pass to a module whenever it is loaded
 - Aliases to represent a module name (other than network)
 - Commands to execute when a module is loaded or unloaded
 - "Blacklisted" modules do not auto-load when hot-plugged

The **lsmod** utility, which gets its information from `/proc/modules`, provides a list of all loaded modules, along with the corresponding size, usage count, any which modules require it (if any).

```
[root@host ~]# lsmod | grep usb_storage
usb_storage      72609  1
```

Modules may be loaded or unloaded using the **insmod**, **rmmod**, or **modprobe** utilities. Unlike **insmod**, **modprobe** will automatically load dependencies and locate the required modules and files by searching `/lib/modules/$(uname -r)/`. Loading a module is as easy as:

```
[root@host ~]# modprobe usb_storage
```

To remove a currently loaded module, use **modprobe -r**:

```
[root@host ~]# modprobe -r usb_storage
```

Note that a module may only be removed if it is not currently in use. A module *can* be force removed using **rmmod --force module_name** if the kernel was compiled with the `CONFIG_MODULE_FORCE_UNLOAD` option, but it is extremely dangerous to do so.

The **modinfo** command can be passed a module name or filename. It will display the associated information, such as the author's name, license, description, module version, dependencies, parameters, etc.

```
[root@host ~]# modinfo usb_storage
filename:          /lib/modules/2.6.18-1.2747.el5/kernel/drivers/usb/storage/usb-storage.ko
license:           GPL
description:       USB Mass Storage driver for Linux
author:            Matthew Dharm
srcversion:        2AA118E385318B2C39E10D3
```

```
depends:      scsi_mod
vermagic:    2.6.18-1.2747.el5 SMP 686 REGPARM 4KSTACKS gcc-4.1
parm:        delay_use:seconds to delay before using a new device (uint)
```

The `/etc/modprobe.conf` configuration file contains persistent settings added at install time and is the usual location for local customizations. `/etc/modprobe.d/` extends the configuration with system defaults that usually don't need to be edited. One exception in that drop directory is `/etc/modprobe.d/blacklist` which describes modules that should never be loaded on the system. This capability is useful when there is more than one device driver that could be used for a device, but a specific one is required. The following line entries are typical of those found in `/etc/modprobe.conf`.

```
alias eth0 airo
alias snd-card-0 snd-intel8x0
options snd-intel8x0 index=0
```

Beginning with RHEL 5.0, network modules are named according to the `HWADDR` and `DEVICE` variables in `/etc/sysconfig/network-scripts/ifcfg-name`. This is accomplished via **udev** as defined in the following rule files:

```
[root@host ~]# grep net /etc/udev/rules.d/*
```

#lspci



- Catalog of devices recognized by the system
 - `/etc/sysconfig/hwconf`
 - **hwbrowser**
 - **hal-device-manager** (hal-gnome package)
- Device information
 - `/sys`
 - `udevinfo -ap $(udevinfo -q path -n /dev/sda)`

hal - hardware
abstraction
layer

udev



- **Typical #include's:**
 - `linux/module.h`: required by all modules
 - `linux/kernel.h`: required for `printk()` loglevel specification
 - `linux/init.h`: required for `module_init` and `module_exit` macros
- **Macros to call initialization and cleanup functions**
 - Functions must be defined before calling macros
 - `module_init(fxn_init);`
 - Called only once, at module load time
 - Initialization function often used with `__init`
 - `module_exit(fxn_exit);`
 - Called only once, at module unload time
 - Without it, a module cannot be unloaded
 - Cleanup function often used with `__exit`



- Defined in `include/linux/module.h`
- `MODULE_ALIAS(alias)`
 - From userspace, this module can also be referred to as *alias*
- `MODULE_LICENSE(license)`
 - Several GPL and dual-licensed components
 - For users wanting evidence their setup is free
 - For identifying proprietary modules in bug reports
- `MODULE_AUTHOR(author)`
 - Typically: `Name <e-mail>[, Name <e-mail>]*[ and Name <e-mail>]`
- `MODULE_DESCRIPTION(description)`
 - What the module does
- `MODULE_PARM_DESC(parm, desc)`
 - One per parameter, describing its use and/or permitted values
- `MODULE_VERSION(version)`
 - Format: `[<epoch>:]<version>[-<extra-version>]`
- `MODULE_FIRMWARE(firmware)`
 - Name of the firmware file needed by the module (one line per firmware file)

See `include/linux/module.h` for more information regarding these functions and specific license text strings that can be used with the `MODULE_LICENSE` macro to indicate a free software module:

"GPL"	GNU Public License v2 or later
"GPL v2"	GNU Public License v2
"GPL and additional rights"	GNU Public License v2 rights and more
"Dual BSD/GPL"	GNU Public License v2 or BSD license choice
"Dual MIT/GPL"	GNU Public License v2 or MIT license choice
"Dual MPL/GPL"	GNU Public License v2 or Mozilla license choice

The "Proprietary" license string can also be used to specifically identify the module as non-free.

Note that any module loaded into the kernel without the license specification macro or one specifying a non-free license string will *taint* the kernel. If a kernel has been tainted since boot time, the value in `/proc/sys/kernel/tainted` will change to a non-zero value according to the following bit-shifts in `include/linux/kernel.h`:

```
#define TAIN_T_PROPRIETARY_MODULE      (1<<0)
#define TAIN_T_FORCED_MODULE           (1<<1)
#define TAIN_T_UNSAFE_SMP              (1<<2)
#define TAIN_T_FORCED_RMMOD            (1<<3)
#define TAIN_T_MACHINE_CHECK           (1<<4)
#define TAIN_T_BAD_PAGE                 (1<<5)
#define TAIN_T_UNSIGNED_MODULE         (1<<6)
```

For example, the loading of an unsigned non-free module will result in a value of 65 in `/proc/sys/kernel/tainted`.

The tainted value is not reset until the next boot of an untainted kernel. A tainted kernel may affect the support options of your kernel. When seeking assistance from Red Hat Global Support, they may require you to reproduce any problems without the presence of the module that tainted the kernel.

For `MODULE_VERSION(version)`, the version is broken up into three pieces: the *epoch*, an optional unsigned integer which allows versions to start fresh, the *version*, an alphanumeric string (may contain the period character), and *extraversion*, optionally used for local customizations (e.g. `test1`).



- The kernel (*kernel space*) does not have access to C library functions (*user space*)
- `printk()` is the kernel's version of `printf()`
- Doesn't flush until a trailing newline is provided
- Behaves similar to `printf()` with some exceptions:
 - No floating-point support
 - Can specify message severity by associating with a `loglevel` macro
 - Outputs to `/proc/kmsg`
 - Tunable parameters
- Example:

```
printk("Test message number %d\n", msgnum);
```

While a typical user-space application can call functions it doesn't define by linking to libraries like `libc`, modules are only linked to the kernel and have access only to the kernel's public symbols (functions and variables), including `printk()`. `printk()` is the version of `printf()` that is exported by the kernel to modules in kernel space.

There is no floating-point support in `printk()`, or anywhere in the kernel, for that matter (performance issue).



- **klogd** listens for messages on `/proc/kmsg` (`printk()` ring buffer)
- **klogd** is managed by **syslog** service script
 - `/etc/sysconfig/syslog`
 - `[root@host ~]# service syslog status`
- If **klogd** is running:
 - Messages are delivered to **syslogd** and processed by `/etc/syslog.conf`
 - By default, messages are appended to `/var/log/messages`
 - Can uncomment and customize kernel line in configuration file:
 - `kern.* /dev/console`
- If **klogd** is not running:
 - **dmesg** can read `/proc/kmsg` directly from user space
 - `[root@host ~]# cat /proc/kmsg`
- In the case of contiguous identical output, **klogd** prints the first line and the count of repeats

The default size of the `printk()` ring buffer is set in the kernel configuration file, but can also be configured as a kernel parameter at boot time (`log_buf_len=n`). See `Documentation/kernel-parameters.txt` for more information.



- `printk()` messages are assigned severity via pre-defined loglevels
- Example:

```
printk(KERN_INFO "My message\n");
```

- Message loglevel, if unspecified, defaults to `KERN_WARN`
 - Defined in `kernel/printk.c`
- Console loglevel
 - If message loglevel is less than console loglevel, the message is sent to the console
 - `klogd` operates independent of the `console_loglevel` value
- `kernel.printk` holds four integers (default values: 6 4 1 7):
 1. The current console loglevel (msgs with higher severity than this will be printed to the console)
 2. Default message loglevel if not specified in `printk()`
 3. Minimum message loglevel (highest severity) that can be used
 4. The default console loglevel at boot time
- The console loglevel can also be modified with:
 - `-c n` option to `klogd` in `/etc/sysconfig/syslog`
 - Kernel boot parameter: `loglevel=[0-7]`
 - `syslog(2)`

Notice there is no comma between the message priority and the message in the above example. Adding a comma is a common mistake for the beginning module programmer, but fortunately the compiler will catch this mistake. There is no comma because at compile time the loglevel macro expands to a string and the message is appended to it.

Each loglevel macro corresponds to an integer between 0-7 enclosed in angle bracket characters. See `include/linux/kernel.h` for more information.

`kernel.printk` values can be temporarily changed using the command (replace the values with your customized ones):

```
[root@host ~]# echo "6 4 1 7" > /proc/sys/kernel/printk
```

Note that using a smaller set of values in the `echo` command will replace values at the front-end of the list. To change just the first value (in this case, so that *all* messages will appear on the console), the following command could be used:

```
[root@host ~]# echo 8 > /proc/sys/kernel/printk
```

For changes that persist a reboot, place the following line (replace the values with your customized ones) in `/etc/sysctl.conf`:

```
kernel.printk="6 4 1 7"
```

The following table lists the various loglevels and their usual meaning.

Loglevel Macro	Text Value	Description
<code>KERN_EMERG</code>	<0>	System is unusable (usually precedes a crash)
<code>KERN_ALERT</code>	<1>	Action is required immediately
<code>KERN_CRIT</code>	<2>	Critical conditions (usually hardware or software failures)

KERN_ERR	<3>	An error condition has occurred (usually hardware related)
KERN_WARN	<4>	A warning that, by itself, may not indicate something critical
KERN_NOTICE	<5>	A normal, but noteworthy event (many security messages)
KERN_INFO	<6>	Purely informational (often startup or device discovery notices)
KERN_DEBUG	<7>	Used for debugging



- Placing a limit on the number of identical messages prevents:
 - Denial-of-service attack
 - Overflowing the system log file
 - Flooding a slow console device, causing the system to become unresponsive
- `int printk_ratelimit(void);`
 - Should be called if a message could be repeated many times
 - Tracks how many messages are sent
 - When a threshold is exceeded, returns 0
 - Example:

```
if (printk_ratelimit())
    printk(KERN_WARNING "Alarm is ringing\n");
```

- Tunables:
 - `kernel.printk_ratelimit`: minimum time (in jiffies) between messages
 - `kernel.printk_burst`: number of messages allowed before threshold reached
- Rules of thumb:
 - On a production system, don't print anything during normal operation
 - Print a message only for an event requiring attention

The different log levels that can be used for the `loglevel` boot time parameter are specified in `Documentation/kernel-parameters.txt`

In the above example, so long as the threshold has not been reached, `printk_ratelimit()` returns a non-zero number and the message is printed. As soon as the threshold is reached, the message is silently dropped.

The tunable values can be set temporarily using a command similar to:

```
[root@host ~]# echo 10 > /proc/sys/kernel/printk_burst
```

or made to persist a reboot by entering a line like the following in `/etc/sysctl.conf`:

```
kernel.printk_burst=10
```

sysctl



```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init init_function(void)
{
    printk(KERN_INFO "Example module loading\n");
    return 0;
}

static void __exit exit_function(void)
{
    printk(KERN_INFO "Example module unloading\n");
}

module_init(init_function);
module_exit(exit_function);
MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Joe Student <student@example.com>");
MODULE_DESCRIPTION("Example module template");
```

The `__init` macro places the function into the `.init.text` section, which will later be freed from memory after the first call by `free_init()`. There is a corresponding macro, `__initdata`, which is used to place variables into the `.init.data` section.

The `__exit` and `__exitdata` macros are essentially the same, but they are placed in `.exit.text` and `.exit.data` (respectively), and they are never called if the module was compiled statically into the kernel (instead of as a dynamically loadable module).



1. Install kernel-devel package to match current kernel version

- Populates symbolic link at `/lib/modules/$(uname -r)/build`
- Provides kernel headers and makefiles sufficient to build modules
- `[root@host ~]# yum -y install kernel-devel`

2. Create a Makefile in the directory of the module source code

```
obj-m := module_name.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

3. Compile the module

- `[root@host ~]# make`

4. Load the module into the kernel

- `[root@host ~]# insmod module_name.ko`

5. Verify

- `[root@host ~]# lsmod | grep module_name`

Note the 2.6 version of the kernel introduced a new naming convention for kernel modules, using the `.ko` extension (kernel object) rather than the `.o` (object) extension used in previous releases.

The module could also have been compiled using a single-line Makefile:

```
obj-m := module_name.o
```

and the command:

```
[root@host ~]# make -C /lib/modules/$(uname -r)/build M=$PWD modules
```

If there are several object files that are used to produce the module, the Makefile can be extended as follows:

```
obj-m += mod1.o
obj-m += mod2.o
mod2-objs := mod2_1.o mod2_2.o
```



1. Extend our Makefile with the following lines:

```
install:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) ✓
modules_install
```

2. Installs to /lib/modules/\$(uname -r)/extra
3. Update modules.dep so **modprobe** can now be used

- [root@host ~]# depmod -a

4. Verify

- [root@host ~]# modprobe module_name; lsmod | grep module_name



- Within each kernel Makefile directory there should be a Kconfig
- Provides a standard means for specifying if and how the module should be compiled and included with the kernel
- Kconfig contains the following:
 - Description of the module and why might it be needed
 - Default compilation method (dynamic module, static, or not at all)
 - Dependencies among compilation parameters
- Example Makefile used with Kconfig:

```
obj-$(CONFIG_MOD1) += mod1.o
obj-$(CONFIG_MOD2) += mod2.o
mod2-objs := mod2_1.o mod2_2.o
```

- Corresponding example Kconfig:

```
menu "Networking"
config NET
    bool "Networking support"
if NET
menu "Test Modules"
config MOD1
    bool "MOD1 Support"
    default y
    help
        Text displayed when help is selected for MOD1.
config MOD2
    tristate "MOD2 Support"
    default n
    help
        Text displayed when help is selected for MOD2.
endif
```

The above example specifies that the MOD1 and MOD2 support are buried under two levels of menus. First, under the Networking menu is a another menu selection labeled Networking Support. If Networking Support is enabled (Boolean), then the Test Modules menu will appear. Within the Test Modules menu, MOD1 Support is a selectable Boolean that is enabled by default, and MOD2 Support is a tristate (y=static, m=module, n=no) selection. If the character h is selected when either MOD1 Support or MOD2 Support is highlighted the corresponding help text will be displayed.

For a good example of a real Kconfig and its features, see the file net/Kconfig.



- Module parameters are declared with a macro
 - See: `include/linux/moduleparam.h`
 - `perm` values defined in `include/linux/stat.h`
 - Macro should be used outside of any function
- Commonly used macros:
 - `module_param(name, type, perm);`
 - `module_param_named(name, value, type, perm);`
 - `module_param_string(name, string, len, perm);`
 - `module_param_array(name, type, numparam, perm);`
- Values can be re-assigned at load time or in configuration file
 - `[root@host ~]# modprobe module_name data=10 text="xyz"`
 - `/etc/modprobe.conf`
 - `options module_name data=10 text="xyz"`

The `module_param` macro is the most straight-forward way of defining a parameter for your module. There are other macros that may be more specific to your needs defined in `include/linux/moduleparam.h`. For example, `module_param_named(name, value, type, perm)` (where `name` can be a string that describes the parameter, and different from the assigned value) `module_param_string(name, string, len, perm)` (which passes a string directly into a char array), and `module_param_array(name, type, numparam, perm)` (which is used to pass an array of values via a single parameter, e.g. `data=1,2,3`).

The parameter value can be assigned at load time (**`insmod`** or **`modprobe`**) or in the `/etc/modprobe.conf` file for more persistent settings.

The module parameter `type` is any of: `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `charp`, `bool` or `invbool`. A custom type can be used (e.g. `type xyz`) if `param_get_xyz`, `param_set_xyz` and `param_check_xyz` are defined. See `include/linux/moduleparam.h` for more information.

The module parameter `perm` uses values defined in `include/linux/stat.h`. The `perm` value defines whether the parameter is presented as `/sys/module/module_name/parameter/parameter_name` in the sysfs structure, and if so, who can access the module parameter and with what permissions. If `perm` is set to 0, no sysfs entry is created. If set to `S_IRUGO|S_IWUSR`, it will have world/other read permissions but only the root user can change it (0644, root.root).

Note: when a parameter is modified in sysfs, the parameter value changes immediately in the module it is associated with, but the module isn't notified of the change in any way.



```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
static int data = 5;
module_param(data, int, S_IRUGO|S_IWUSR);

static int __init init_function(void)
{
    printk(KERN_INFO "Loading example module\n");
    printk(KERN_INFO "Parameter (init): data=%d\n", data);
    return 0;
}
static void __exit exit_function(void)
{
    printk(KERN_INFO "Unloading example module\n");
    printk(KERN_INFO "Parameter (exit): data=%d\n", data);
}
module_init(init_function);
module_exit(exit_function);
MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Joe Student <student@example.com>");
MODULE_DESCRIPTION("Example module w. parameter");
MODULE_PARM_DESC(data, "Dummy Data (default=5)");
```

"Test drive":

```
[root@host ~]# make -C /lib/modules/$(uname -r)/build M=$(PWD) modules_install
[root@host ~]# depmod -a
[root@host ~]# modinfo module_name
filename:        /lib/modules/2.6.18-53.1.14.el5/extra/module_name.ko
description:     Kernel module with parameter
author:          Student <student@example.com>
license:         GPL v2
srcversion:      626B95617F289EE3B787E4B
depends:
vermagic:        2.6.18-53.1.14.el5 SMP mod_unload 686 REGPARM 4KSTACKS
gcc-4.1
parm:            data: Dummy Data (default=5) (int)

[root@host ~]# modprobe module_name; tail -1 /var/log/messages
May 19 17:12:00 dhcp235-161 kernel: Parameter (init): data=5

[root@host ~]# ls -l /sys/module/module_name/parameters/data
-rw-r--r-- 1 root root 4096 May 19 17:12 /sys/module/module_name/parameter
s/data
```



```
[root@host ~]# cat /sys/module/module_name/parameters/data
5
[root@host ~]# echo 10 > /sys/module/module_name/parameters/data
[root@host ~]# cat /sys/module/module_name/parameters/data
10
[root@host ~]# modprobe -r module_name; tail -1 /var/log/messages
May 19 17:13:21 dhcp235-161 kernel: Parameter (exit): data=10
[root@host ~]# modprobe module_name data=20; tail -1 /var/log/messages
May 19 17:26:44 dhcp235-161 kernel: Parameter (exit): data=20
```

Other examples of using module parameters:

```
static char *test = "Hello";
module_param(test, charp, S_IRUGO);

module_param_named(text, "abc", charp, S_IRUGO|S_IWUSR);

static int con_opt = 1;
module_param_named(con, con_opt, int, 0444);
MODULE_PARM_DESC(con, "Console. 0=off, 1=on (default)");

#define MAX_OPTIONS_SIZE 256
module_param_string(opts, boot_opts, MAX_OPTIONS_SIZE, 0);
MODULE_PARM_DESC(opts, "Options. \"opts=a:b:c\"");

static char vid_mode[32] = "800x600:16@72";
module_param_string(vm, vid_mode, sizeof(vid_mode), 0444);
MODULE_PARM_DESC(vm, "VideoMode (<x>x<y>[:<bpp>] [@<Hz>])");
```



End of Lecture 4

- Questions and Answers
- Summary
 - Loading and unloading kernel modules
 - How to write and implement a simple kernel module







Lab 4.1: Building and Loading a Simple Module

Deliverable: A simple kernel module and an understanding of how to compile and load it

Instructions:

1. First, make sure that you are currently using the latest version of the kernel (reboot into it if you have to). Make sure this version of the kernel will be the default upon startup. ✓
2. If not already installed, make sure the `kernel-devel` package for your current kernel version is installed. If not, install it now. ✓
3. What is the difference in the location of the files installed by the `kernel-devel` package versus a compile of the kernel from the source RPM package?

-
4. Where does the link `/lib/modules/kernel-version/build` point to?

/usr/src/redhat/BUILD/kernel-2.6.18/linux-2.6.18.1386

5. Log in as the `student` user on your machine (username: `student`, password: `student`) and create a working directory named `kmod`.

There is a template available at `instructor:/var/ftp/pub/rhd361-materials/template.c`.

6. In the working directory you just created, create a simple GPL-licensed kernel module that prints the message "Hello World". Create a proper `Makefile` to compile it, and then compile it. ✓
7. As the `root` user (user `student` does not have sufficient privileges, by default), load the module. Where did the output go? ✓

-
8. Verify your module is actually loaded and view the licensing and other information you built into it. Unload the module when you are done. ✓

rhd-361 solutions

instructor. sample.1cm

Lab 4.2: Deploying a Kernel Module

Deliverable: A kernel module that can be loaded and unloaded using **modprobe**

Instructions:

1. Notice the **insmod** command used in the previous step had to specify a fully qualified pathname to the module in order to find and load it.
✓
- ✓ Perform all steps necessary so you can load your module using the **modprobe** command from anywhere in the filesystem without specifying a fully qualified pathname to your module.
- ✓ Unload the module when you are done testing.

Lab 4.3: Module Initialization and Exit Functions

Deliverable: A deeper understanding of kernel module initialization and exit functions

Instructions:

1. What happens if you try to load the module twice using **modprobe**? Using **insmod**?

insmod - 1st time works, modprobe - no error, but no duplicates

2. Create a new version of the kernel module that does not have an initialization function. Can you still load the module?

3. Create a new version of the kernel module that does not have an exit function. Can you load the module? Can you unload the module?
-

Lab 4.4: Implementing Parameters in a Kernel Module

Deliverable: A kernel module with a variable that can be queried and altered

Instructions:

1. Edit your "hello world" module so that it has a variable named `data` whose value is initially set to 100, but can be altered from user space via a `/sys` entry.

Lab 4.1 Solutions

1. First, make sure that you are currently using the latest version of the kernel (reboot into it if you have to). Make sure this version of the kernel will be the default upon startup.

```
[root@host ~]# yum list kernel
[root@host ~]# uname -r
[root@host ~]# cat /etc/grub.conf
```

(Make sure the default parameter in `/etc/grub.conf` points to the correct kernel version.)

2. If not already installed, make sure the `kernel-devel` package for your current kernel version is installed. If not, install it now.

```
[root@host ~]# yum list kernel-devel
[root@host ~]# yum -y install kernel-devel
```

3. What is the difference in the location of the files installed by the `kernel-devel` package versus a compile of the kernel from the source RPM package?

```
[root@host ~]# rpm -ql kernel-devel | less
```

The `kernel-devel` package installs files in the path `/usr/src/kernels/version`, whereas the source RPM uses the `/usr/src/redhat/BUILD/kernel-version/linux-version` directory for compiling the pieces needed to compile a kernel module.

4. Where does the link `/lib/modules/kernel-version/build` point to?

```
[root@host ~]# ll /lib/modules/version/build
lrwxrwxrwx 1 root root 48 Mar 17 06:27 ✓
/lib/modules/version/build -> ✓
../../../../usr/src/kernelsversion/
```

5. Log in as the student user on your machine (username: student, password: student) and create a working directory named `kmod`.

There is a template available at `instructor:/var/ftp/pub/rhd361-materials/template.c`.

If already logged in as the root user:

```
[root@host ~]# su - student
[student@host ~]$ mkdir kmod
```

Optional:

```
cp ✓
/net/instructor/var/ftp/pub/rhd361-materials/temp ✓
late.c ~student/kmod
```

6. A solution for this exercise is located on the instructor machine at `/var/ftp/pub/rhd361-solutions/04-modules`. The source code for this specific problem is called `helloworld.c` and a Makefile that compiles the solution kernel module is provided.

Compile the kernel module:

```
[student@host ~]$ make
make -C /lib/modules/version/build ✓
M=/home/student/kmod modules
make[1]: Entering directory `/usr/src/kernels/version'
CC [M] /home/student/kmod/helloworld.o
Building modules, stage 2.
MODPOST
CC /home/student/kmod/helloworld.mod.o
LD [M] /home/student/kmod/helloworld.ko
make[1]: Leaving directory `/usr/src/kernels/version' ✓
```

7. As the root user (user `student` does not have sufficient privileges, by default), load the module. Where did the output go?

```
[root@host ~]# insmod ~student/kmod/helloworld.ko
[root@host ~]# grep Hello /var/log/messages
[root@host ~]# dmesg | grep Hello
```

8. Verify your module is actually loaded and view the licensing and other information you built into it. Unload the module when you are done.

```
[root@host ~]# lsmod | head
[root@host ~]# modinfo ✓
~student/kmod/helloworld.ko
filename:          helloworld.ko
description:       Hello world kernel module
author:            Student <student@example.com>
license:           GPL v2
srcversion:        99A693C033409042D73595C
depends:
vermagic:          version SMP mod_unload 686 ✓
REGPARM 4KSTACKS gcc-4.1
[root@host ~]# rmmod helloworld
```

Lab 4.2 Solutions

1. Perform all steps necessary so you can load your module using the **modprobe** command from anywhere in the filesystem without specifying a fully qualified pathname to your module.

Begin by copying your module to the standard location for custom modules (actually, anywhere under `/lib/modules/version/` will do):

```
[root@host ~]# cp ~student/kmod/helloworld.ko \
/lib/modules/version/extra/
[root@host ~]# depmod -ae
```

Verify it was mapped into the module dependency file for use by the **modprobe** command:

```
[root@host ~]# grep helloworld /lib/modules/versi
on/modules.dep
/lib/modules/version/extra/helloworld.ko:
```

Verify that it now works with **modprobe**:

```
[root@host ~]# modprobe helloworld
[root@host ~]# lsmod | grep helloworld
helloworld                5760  0
```

Unload the module when you are done testing.

```
[root@host ~]# modprobe -r helloworld
[root@host ~]# lsmod | grep helloworld
```


Lab 4.3 Solutions

1. What happens if you try to load the module twice using **modprobe**? Using **insmod**?

```
[root@host ~]# modprobe helloworld
[root@host ~]# lsmod | grep helloworld
helloworld                5760  0
[root@host ~]# modprobe helloworld
[root@host ~]# lsmod | grep helloworld
helloworld                5760  0
[root@host ~]# insmod /lib/modules/version/extra/ ✓
helloworld.ko
insmod: error inserting '/lib/modules/version/ext ✓
ra/helloworld.ko': -1 File exists
```

2. Create a new version of the kernel module that does not have an initialization function. Can you still load the module?

Answer: yes

3. Create a new version of the kernel module that does not have an exit function. Can you load the module? Can you unload the module?

Answer: You won't be able to unload the module using **rmmod** or **modprobe**. What must you do to unload it? - reboot!

Lab 4.4 Solutions

1. A solution for this exercise is located on the instructor machine at `/var/ftp/pub/rhd361-solutions/04-modules`. The source code for this specific problem is called `helloworld2.c` and a Makefile that compiles the solution kernel module is provided.

After compiling the module, query its information:

```
[root@host ~]# make
...output omitted...
[root@host ~]# modinfo helloworld2.ko
filename:          helloworld2.ko
description:       Kernel module with parameter
author:           Student <student@example.com>
license:          GPL v2
...output omitted...
parm:              data: Dummy Data (default=100)
                   (int)
```

Load the module and view its output:

```
[root@host ~]# insmod helloworld2.ko
[root@host ~]# dmesg | tail
...output omitted...
loading custom RHD361 module.....
Parameter (init): data=100
```

Query its data value directly via `/sys`:

```
[root@host ~]# ls -l
/sys/module/helloworld2/parameters/data
-rw-r--r-- 1 root root 4096 Mar 17 08:38
/sys/module/helloworld2/parameters/data
[root@host ~]# cat
/sys/module/helloworld2/parameters/data
100
```

(Note: Who, in this case, has the ability to view the module's data? Edit it? These are considerations when making your own module parameters.)

Now alter the module's data value:

```
[root@host ~]# echo 1 >
/sys/module/helloworld2/parameters/data
[root@host ~]# cat
/sys/module/helloworld2/parameters/data
1
```

Unload the module and view what its data value is upon exit:

```
[root@host ~]# rmmod helloworld2
```

```
[root@host ~]# dmesg | tail
...output omitted...
Unloading custom module....
Parameter (exit): data=1
```






Lecture 5

Kernel API Overview

Upon completion of this unit, you should be able to:

- Describe the main structure used to characterize a process and its resources (the `task_struct`)
- Use memory allocation functions to allocate memory for the kernel and for your kernel modules.
- Use the `copy_{from,to}` functions to transfer data from user space to kernel space.
- Understand other mechanisms of transferring data from user to kernel space.

student@example.com



- Modern OS's allow multiple users to share a machine's resources
- Users are allowed to run multiple tasks
- The kernel must:
 - Protect each task from other tasks
 - Allow each task its share of the processor resource
- To manage multitasking, the kernel uses a data structure to keep track of each task
 - Memory
 - File descriptors
 - Pending signals
 - PID
 - ...
- The data structure used is the *process descriptor*
- Each task needs its own private *stack* of information



- Upon entering `main()`:
 - Exit address
 - Command-line arguments
 - Environment variables
- During execution of `main()`:
 - Function parameters and return addresses
 - Storage locations for automatic variables

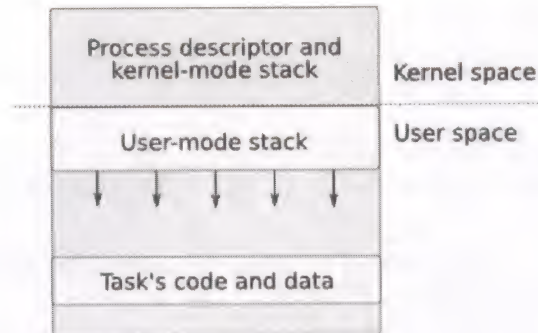


- User mode processes enter kernel mode when:
 - A system call is made
 - Arrival of an interrupt
 - An exception occurs
- The kernel mode switch not only escalates privileges, but a stack-area switch
 - Each task, in addition to its code and data sections, will have a stack in both kernel and user space

The kernel timer is a guaranteed source of interrupts for the user mode process. An example of an exception might be a divide-by-zero error in the task.

A stack area switch is necessary when entering kernel mode for stability and security reasons (prevent access to privileged data areas).

During execution of kernel functions, the kernel stack will contain function parameter, return addresses, and storage locations for automatic variables.



- Each task has a kernel space process descriptor
- Linux uses a task's kernel stack to store thread information
- This thread information includes a pointer to the task's process descriptor data structure



- Instance of a program in execution
- Has address space (stack and heap), program code (text), file descriptors, signal handlers
- Each process has a single `task_struct`
- All resources are defined within `task_struct` data structure (defined in `linux/sched.h`)
- Most kernel references to a process refer to the address of the `task_struct` (known as the process descriptor pointer)
- `current` macro returns the process descriptor pointer of the process currently running on the processor
 - `current->pid`



- For each process, the kernel stores two other data structures in a single 8kB page (4kb for x86) aligned on a multiple of 2^{13}
- The `thread_info` structure points to a process task struct through `*task`
- The kernel stack starts at the top of the page and increases downwards
- The stack pointer (`%esp`) always points to the bottom of the stack

We can define structures created by the following union:

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[2048];
}

struct thread_info {
    28  struct task_struct    *task;           /* main task structure */
    29  struct exec_domain    *exec_domain;    /* execution domain */
    30  unsigned long          flags;           /* low level flags */
    31  unsigned long          status;          /* thread-synchronous flags */
    32  __u32                  cpu;            /* current CPU */
    33  int                    preempt_count; /* 0 => preemptable,
    <0 => BUG */
    36  mm_segment_t           addr_limit;      /* thread address space:
    37                                     0-0xBFFFFFFF for user-thread
    38                                     0-0xFFFFFFFF for kernel-thread
    39                                     */
    40  void                    *sysenter_return;
    41  struct restart_block    restart_block;
    42
    43  unsigned long            previous_esp; /* ESP of the previous stack*/
    46  __u8                    supervisor_stack[0];
    47};
```

With this structure, it is very efficient for the kernel to get the current task struct on a processor. The `%esp` register points to the bottom of the current kernel stack. The `thread_info` structure is always at offset 0, and can be obtained using the `current_thread_info` macro. We can then reference the task field.

Thus, the current macro becomes `current_thread_info(%esp) -> task`.



- `pid` - The process ID
- `tgid` - Provides the thread group ID
 - For heavyweight processes, this is the same as the `pid`
 - For lightweight processes (or threads), this is the `pid` of the thread group leader (which is usually the first thread)
- For POSIX compliance, `getpid()` returns `tgid`
- For efficiency, the kernel stores a mapping from process descriptor pointers to `pids` using hash functions



- Task state bitmask:

- R • TASK_RUNNING: process either running or waiting to run
- S • TASK_INTERRUPTIBLE: process sleeping (possibly waiting on input), but interruptible by signals
- D • TASK_UNINTERRUPTIBLE: same as above but not interruptible by signals
- TASK_STOPPED: process stopped - after receiving a SIGSTOP

- Task exit_state bitmask:

- Z • EXIT_ZOMBIE: child process with unreaped exit status
- EXIT_DEAD: process waiting to be cleaned up by kernel

- Task flags bitmask : process being created, shut down, killed by signal etc.

Notes: The following few slides will have most (but not all) of the most important elements in the task_struct structure.

```
struct task_struct {
    volatile long state;      /* -1 unrunnable, 0 runnable, >0 stopped */
    unsigned int flags;      /* per process flags, defined below */
    int exit_state;
    int exit_code, exit_signal;
```

Values for flags:

```
#define PF_ALIGNWARN      0x00000001    /* Print alignment warning msgs */
#define PF_STARTING      0x00000002    /* being created */
#define PF_EXITING      0x00000004    /* getting shut down */
#define PF_EXITPIDONE      0x00000008    /* pi exit done on shut down */
#define PF_FORKNOEXEC      0x00000040    /* forked but didn't exec */
#define PF_SUPERPRIV      0x00000100    /* used super-user privileges */
#define PF_DUMPCORE      0x00000200    /* dumped core */
#define PF_SIGNALED      0x00000400    /* killed by a signal */
#define PF_MEMALLOC      0x00000800    /* Allocating memory */
#define PF_FLUSHER      0x00001000    /* responsible for disk writeback */
#define PF_USED_MATH      0x00002000    /* if unset the fpu must be
    initialized before use */
#define PF_NOFREEZE      0x00008000    /* this thread should not be frozen
    */
#define PF_FROZEN      0x00010000    /* frozen for system suspend */
#define PF_FSTRANS      0x00020000    /* inside a filesystem transaction */
#define PF_KSWAPD      0x00040000    /* I am kswapd */
#define PF_SWAPOFF      0x00080000    /* I am in swapoff */
#define PF_LESS_THROTTLE 0x00100000    /* Throttle me less: I clean memory
    */
#define PF_BORROWED_MM      0x00200000    /* I am a kthread doing use_mm */
#define PF_RANDOMIZE      0x00400000    /* randomize virtual address space */
#define PF_SWAPWRITE      0x00800000    /* Allowed to write to swap */
#define PF_SPREAD_PAGE      0x01000000    /* Spread page cache over cpuset */
#define PF_SPREAD_SLAB      0x02000000    /* Spread some slab caches over
    cpuset */
```

```
#define PF_MEMPOLICY      0x10000000 /* Non-default NUMA mempolicy */
#define PF_MUTEX_TESTER  0x20000000 /* Thread belongs to the rt mutex tester */
#define PF_FREEZER_SKIP  0x40000000 /* Freezer should not count it as freezeable */
```




- Priority of a process:
 - prio
 - static_prio
 - normal_prio
 - rt_priority
- CPU affinity (mask of CPUs this process can be scheduled on)
 - cpus_allowed
- Scheduling policy and statistics:
 - SCHED_NORMAL
 - SCHED_FIFO
 - SCHED_RR
 - SCHED_BATCH
 - sched_info
- Currently runnable processes:
 - run_list
- Circular doubly-linked list of all task_structs:
 - tasks

The following are some of the key fields defined in the `task_struct` structure:

```
int oncpu;
int prio, static_prio, normal_prio;
struct list_head run_list;
struct sched_class *sched_class;
struct sched_entity se;
unsigned short ioprio;
unsigned int policy;
cpumask_t cpus_allowed;
unsigned int time_slice;
struct sched_info sched_info; (statistics)
unsigned int rt_priority;
struct list_head tasks;
```



- `#include <linux/list.h>`
 - `struct list_head { struct list_head *next, *prev; }`
- `LIST_HEAD()`, `LIST_HEAD_INIT()`
- `list_add()`, `list_del()`, ...
- `list_entry()`
 - Only component with custom knowledge

The Linux kernel provides a doubly linked list API which allows linked lists to be managed generically, with the "local knowledge" isolated to a single `list_entry()` macro. The implementation relies on a `struct list_head`, which is merely contains references to next and previous list heads.

A structure which is to be managed on a linked list generally contains an embedded list head, while an additional list head is generally used to anchor the list. (To assemble N structures, N+1 list heads are involved, one for each structure, and one to anchor the list.)

```
LIST_HEAD(name), LIST_HEAD_INIT(struct list_head *ptr)
```

A `struct list_head` which is anchoring a list is often declared and initialized using the `LIST_HEAD()` declaration macro. A dynamically allocated `struct list_head` can be initialized with `LIST_HEAD_INIT()`.

```
list_entry(ptr, type, member)
```

This macro is the only non-generic aspect of the linked list API. The macro will convert a pointer to `struct list_head`, which is the `member` field of a structure `type`, to a pointer to the encapsulating structure.

The `list_entry()` macro performs the same behavior as, and is in fact implemented as, the `container_of()` macro.



- `list_add()`
- `list_add_tail()`
- `list_del()`
- `list_replace()`
- `list_empty()`
- `list_splice()`
- ...
- API does not provide synchronization

```
void list_add(struct list_head *new, struct list_head *head)
```

Attach the *new* list head to the head of the list anchored by *head*.

```
void list_del(struct list_head *entryentry)
```

Remove the *entry* list head from its current double linked list. Note the entry contains enough information to remove the element and repair the list without referencing the list itself.

```
void list_add_tail(struct list_head *new, struct list_head *head)
```

```
void list_replace(struct list_head *old, struct list_head *new)
```

```
int list_empty(const struct list_head *head)
```

```
void list_splice(struct list_head *list, struct list_head *head)
```

These functions, and others, perform generally self evident operations which are fully documented in the `#include <linux/list.h>` header file.

Note the API does not provide synchronization. Presumably, some synchronization structure (mutex or spinlock) is held while performing these operations.



- `list_for_each() { ... }`
- `list_for_each_entry() { ... }`
- `list_for_each_safe() { ... }`

```
list_for_each(struct list_head *pos, struct list_head *head)
```

The `list_for_each()` macro provides the *for clause* for an iteration over the list anchored by `head`, where the `struct list_head *pos` iterator references the “current” element within the iteration code block.

```
list_for_each_entry(pos, head, member)
```

The `list_for_each_entry()` macro iterates over the list anchored by the `struct list_head *head`, with the expectation that the list contains only uniform structures whose embedded `struct list_head` is the *member* field.

```
list_for_each_safe(pos, extra_ptr, head)
```

Care must be taken when removing list members during iterations. In such cases, an additional `struct list_head` pointer must be provided. The pointer is opaquely used to safely store a reference to the next element while removing the current element.

The following examples illustrate these use of these list iteration macros:

```
struct foo { int fooness; struct list_head f_list; ... };
```

```
LIST_HEAD(foo_list);
```

```
struct list_head *ptr, *extra;  
struct foo *fooptr;
```

```
list_for_each(ptr, &foo_list) {  
    fooptr = list_entry(ptr, struct foo, f_list);  
    printk("fooness: %d\n", fooptr->fooness);  
}
```

```
list_for_each_entry(fooptr, &foo_list, f_list) {  
    printk("fooness: %d\n", fooptr->fooness);  
}
```

```
list_for_each_safe(ptr, extra, &foo_list) { list_del(ptr); }
```



- The `tasks` field in each `task_struct` can be used to loop through all current processes
- The `for_each_process()` macro
 - Defined in `linux/sched.h`
 - Iterates through all tasks starting with process 0 (the swapper)

```
#define for_each_process(p) \
    for (p = &init_task; (p = next_task(p)) != &init_task; )
```

The following code snippet from `kernel/cpu.c` shows how to use the `for_each_process()` macro to iterate over all of the processes on the system:

```
static inline void check_for_tasks(int cpu)
{
    struct task_struct *p;

    write_lock_irq(&tasklist_lock);
    for_each_process(p) {
        if (task_cpu(p) == cpu &&
            (!cputime_eq(p->utime, cputime_zero) ||
             !cputime_eq(p->stime, cputime_zero)))
            printk(KERN_WARNING "Task %s (pid = %d) is on cpu %d\
                          (state = %ld, flags = %lx) \n",
                     p->comm, p->pid, cpu, p->state, p->flags);
    }
    write_unlock_irq(&tasklist_lock);
}
```

Notice the use of synchronization functions used with `for_each_process()` since the macro does not perform locking on its own.



- `real_parent`: process that created this process.
- `parent`: same as `parent`, unless a debugger is attached
- `group_leader`: thread group leader, process to whom signals are directed for the thread group

The following fields defined in the `task_struct` structure link processes to their relatives:

```
struct task_struct *real_parent; /* real parent process */
struct task_struct *parent;      /* parent process */
struct list_head children;       /* list of my children */
struct list_head sibling;        /* linkage in my parent's
                                children list */
struct task_struct *group_leader; /* threadgroup leader */
struct list_head thread_group;
u32 parent_exec_id;
u32 self_exec_id;
```




- start and execution times
- page faults
- resident set size

The following fields defined in the `task_struct` structure relate to process statistics:

```
cputime_t utime, stime;
unsigned long nvcsw, nivcsw; /* context switch counts */
struct timespec start_time;      /* monotonic time */
struct timespec real_start_time; /* boot based time */
unsigned long min_flt, maj_flt;
u64 rchar, wchar, syscr, syscw; /* i/o counters(bytes read/
                                written, #syscalls */

struct task_io_accounting ioac;
u64 acct_rss_mem1; /* accumulated rss usage */
u64 acct_vm_mem1; /* accumulated virtual memory usage */
cputime_t acct_stimexpd; /* stime since last update */
```



- `#include <linux/slab.h>`
- `kmalloc(size_t size, gfp_t flags)`
- *flags* can have the value...
 - `GFP_KERNEL` in process context
 - `GFP_ATOMIC` in system context
- `kfree(ptr)`

While memory management is an involved topic, and the Linux kernel provides a variety of different memory allocators for different situations, the easiest and most familiar way to allocate memory in the kernel is with `kmalloc()`.

```
void *kmalloc(size_t size, gfp_t flags)
```

Allocate a memory buffer, returning the address of the buffer, or `NULL` on failure. For our introductory purposes, the flag should be `GFP_KERNEL` in *process context*, or `GFP_ATOMIC` in *system context*, both of which request memory to be drawn from the *normal zone*.

`GFP_KERNEL`: The kernel may perform I/O to free memory, possibly causing the caller to block.

`GFP_ATOMIC`: Fail if memory is not immediately available, instead of blocking the caller.

```
void kfree(const void *ptr)
```

Free memory obtained with `kmalloc()`.

Below is sample code that shows how to allocate and free some kernel memory:

```
#include <linux/slab.h>

char *buffer;

buffer = kmalloc(8192, GFP_KERNEL);
if (!buffer)
    return -ENOMEM;
...

kfree(buffer);
```



- Branch Prediction
 - `#include <linux/compiler.h>`
 - `likely(condition)`
 - `unlikely(condition)`
 - Compiler generates jump instruction for least likely branch

When faced with a condition that has a dominant outcome, a common optimization is to keep the most likely code path local, and generate jumps for less likely code paths. Performing this optimization “manually” often involves using `goto` statements and labels resulting in difficult to follow code.

The Linux kernel provides the `likely()` and `unlikely()` macros, which do not change the logic of a condition, but leverage the GCC compiler's `__builtin_expect()` function which requests the compiler to leave the dominant path local, resulting in more readable code.



- Binding Structures

- `#include <linux/kernel.h>`
- `container_of(ptr, type, member)`
- Macro determines address of encapsulating structure

The easiest way to have two related structures reference one another is by having one structure maintain a pointer to the other, with the two structures residing arbitrarily in memory. When possible, the Linux kernel attempts to keep related structures close in memory by binding them with an encapsulating structure which can be allocated as a unit.

As an example, consider the following `foo` and `baz` structures:

```
struct foo { int flags; atomic_t count; };
struct baz { int state; };

struct bind_foo_baz {
    struct foo s_foo,
    struct baz s_baz,
};

struct foo *pfoo = somehow_obtain_a_foo();
struct baz *pbaz = &container_of(pfoo, struct bind_foo_baz, s_foo)->s_baz;
```

Examination of the `container_of` macro definition in `<linux/kernel.h>` reveals the clever mechanism of determining the desired offset by casting the address 0x0 as the relevant structure. For a “real world” example of a binding structure in the kernel, take a look at the definition and use of the `socket_alloc` structure defined in `<net/sock.h>`.



- OOPS messages
 - Stack trace, register dump, tainted state, and more reported as kernel messages
- `BUG()` and `BUG_ON(condition)`
 - Generate oops
 - Raise architecture appropriate exception (or `panic()`)
 - Possibly recoverable with, for example, context switch
- `panic(msg)`
 - Unrecoverable condition
 - Generate oops and panic message
 - Halt system (often with flashing keyboard LEDs)
 - `/proc/sys/kernel/panic`, `/proc/sys/kernel/panic_on_oops`

On abnormal conditions, the kernel will generate an *oops* message which contains debugging context information including a stack trace, register dump, tainted state, and more. The `die()` function found in `arch/i386/kernel/traps.c` provides an example of the architecture dependent behavior.

`BUG()`, `BUG_ON(condition)`

The `BUG()` and `BUG_ON(condition)` macros generate an architecture appropriate exception. The exception handler will generally generate an oops and, if appropriate, try to recover by slaying the current process (with a `SIG_SEGV`) and scheduling. In system context, the kernel generally *panics*.

`void panic(char *msg)`

Print one final message, and halt the kernel. Often, the keyboard LEDs are left blinking to give visual indication of the abnormal state.

`/proc/sys/kernel/oops - panic`

Configures the kernel to reboot the specified number of seconds after a panic. The default value of 0 implies no reboot is attempted.

`/proc/sys/kernel/panic_on_oops`

When nonzero, the kernel will panic on an oops in *process context* rather than attempt to recover.



End of Lecture 5

- Questions and Answers
- Summary
 - The main structure used to describe the state of a process

Calls to allocate memory for the kernel and how to transfer data to/from user space





Lab 5.1: Querying Process Task Structures

Deliverable: A kernel module which lists the current tasks running on the system

Instructions:

1. Write a module that queries all running tasks on the system and displays the following information for each process in the system log:
 - process ID
 - process state
 - command name

Note this module doesn't need to do anything fancy. The purpose of this lab exercise is to get you started working with existing kernel data structures.

Lab 5.2: Managing Linked Lists

Deliverable: A more sophisticated kernel module which lists the current tasks running on the system

Instructions:

1. Write a module that queries all running tasks on the system and builds a linked list of `pinfo` structures which consist of the following definition:

```
struct pinfo {
    pid_t  pid;
    long   state;
    char   comm[TASK_COMM_LEN];
    struct task_struct *tsk;
    struct list_head p_list;
};
```

The first three fields in the `pinfo` structure correspond to the process PID, state, and command string respectively. The `tsk` field should point to the task structure itself.

Once the list is constructed, traverse the list and print information about each process to the system log and delete the `pinfo` items from your linked list.

The purpose of this lab exercise is to get you started creating and manipulating your own kernel data structures.

Lab 5.1 Solutions

1. Write a module that queries all running tasks on the system and displays the following information for each process in the system log:

- process ID
- process state
- command name

A solution for this exercise is located on the instructor machine at `/var/ftp/pub/rhd361-solutions/05-kernelapi`. The source code for this specific problem is called `listtasks.c`.

Lab 5.2 Solutions

1. A solution for this exercise is located on the instructor machine at `/var/ftp/pub/rhd361-solutions/05-kernelapi`. The source code for this specific problem is called `linkedlist.c`.







Synchronization

Upon completion of this unit, you should be able to:

- Understand the need for synchronization
- Recognize the need for synchronization
- Use Linux kernel tools to provide synchronization
- Know which tool to use in which context



- Data structures must be self consistent
- Code paths require exclusive access...
 - when "breaking" structures
 - when "unbroken" structures are needed
- Such code paths are "critical sections"

Invariants and a Linked List

Even the simplest data structures can be thought of as having *invariants* which must be kept self consistent. The following might be the invariants for a *linked list*.

1. Some pointer references the first element.
2. Each element references the next element.
3. The last element's reference pointer is NULL.

Violating one of these invariants leaves a "broken" linked list.

Inserting an element

Consider an insertion operation which adds a new element to a top of the list, as sketched in the following:

```
new->next = top;  
top = new;
```

In between the two operations, the code path can be interrupted, and forced off the CPU, leaving a "broken" data structure. Or, a concurrent code path on another CPU could be operating on the same data structure.

Whenever a code path needs to "break" a data structure, even temporarily, it must insure that no other code path is relying on that data structure being well formed.

Critical Sections

Branches of code which break and then restore data structures are called *critical sections*. In addition to the linked list above, the following code snippets can be considered "one line" critical sections:

```
stack->data[stack->idx++] = new    /* stack push operation */  
i++                               /* even integer increment !! */
```



- Bound critical sections
- `lock()` and `unlock()` operations
- Advisory - must "play by the rules"
- Simple in concept, hard in implementation
- Often implemented using atomic instructions (`test_and_set`, ...)

Mutual Exclusion Devices (Mutexes)

Data structures are usually protected by bounding relevant critical sections with some variant of a *Mutual Exclusion* device, or *mutex*. Mutexes can be thought of as an integer which supports the following two operations:

```
int m = 1;

lock(m):  while (m != 1) { /* block */ }
          m--;

unlock(m): m++;
```

Mutexes protect data structures by bounding all critical sections associated with the structure. Continuing the example from the previous page, a linked list could be protected by declaring a mutex with the same visibility as the list:

```
struct foo    *top;
struct mutex toplock;

lock(&toplock);
    new->next = top;
    top = new;
unlock(&toplock);
```

If two concurrent code paths attempted to add an element to the linked list, the first would acquire the mutex and proceed. The second would attempt to acquire the mutex, but block until the first released the mutex. The second mutex could then acquire the mutex and continue.

Not just this particular code path must be protected, however. All code paths that either read or manipulate the linked list must first acquire the mutex, and release it once the data structure is repaired.



- Atomic Bit Operations
- Atomic Integers
 - Solve the `i++` problem
- Spinlocks
 - Spin the CPU while blocking
- Mutexes / Semaphores
 - Yield the CPU while blocking

The previous two pages have discussed mutexes as a general concept. The next few pages will look in detail at various implementations of mutexes which are available within the kernel. Each variant is optimized for a particular use case or context.



- Architecture Independent API
- `#include <asm/bitops.h>`
- Basic Manipulations
 - `void set_bit(...), void clear_bit(...), void change_bit(...)`
- Returning Operations
 - `int test_and_set_bit(...), ...`

Bit Operations

At the lowest level, the kernel provides an architecture independent implementation of basic bit operations, defined within the `#include <linux/bitops.h>` header file.

Basic Operations

```
void set_bit(int nr, volatile unsigned long * addr)
```

```
void clear_bit(int nr, volatile unsigned long * addr)
```

```
void change_bit(int nr, volatile unsigned long * addr)
```

These functions perform basic manipulations of a bit at an arbitrarily large offset `nr` from a base address `addr`. They are atomic in the sense that they are guaranteed not to be reordered, and simultaneous operations will each distinctly occur.

Returning Operations

```
int test_and_set_bit(int nr, volatile unsigned long * addr)
```

```
int test_and_clear_bit(int nr, volatile unsigned long * addr)
```

```
int test_and_change_bit(int nr, volatile unsigned long* addr)
```

These functions perform the as above, but return the value of the bit before the operation occurs. They are atomic in the sense that the two operations cannot be interrupted by another code path.

Simple Mutex Implementation

A simple, low level mutex can be implemented using `test_and_clear()` as the locking operation, and `test_and_set()` as the unlocking operation. When locking, if the return value is 1, the lock was successfully obtained. If the return value is 0, the the lock was already obtained.



- Integers supporting atomic increments
- Use case: reference counting
- `#include <asm/atomic.h>`
- Simple Operations
 - `atomic_read(...), atomic_set(...), ...`
- Atomic Operations
 - `atomic_inc(...), atomic_dec(...), ...`

Atomic Integers

```
typedef struct { volatile int counter; } atomic_t
```

If two CPUs were to concurrently increment an integer, the instructions could be interleaved as (simplistically) "fetch", "fetch", "increment", "increment", "put back", "put back", leaving an inconsistent count. Atomic integers, cast as a `atomic_t`, and their associated API are designed as integers which can be safely incremented and decremented.

```
#define ATOMIC_INIT(i) { (i) }  
  
#define atomic_read(v) ((v)->counter)  
  
#define atomic_set(v,i) (((v)->counter) = (i))
```

Simple macros are provided which for initializing, reading, and setting the atomic integer. While these operations are not strictly "atomic" in nature, they provide a consistent API.

```
void atomic_add(int i, atomic_t *v)  
  
void atomic_sub(int i, atomic_t *v)  
  
void atomic_inc(atomic_t *v)  
  
void atomic_dec(atomic_t *v)
```

These operations allow the integer to be safely manipulated, where `atomic_inc(v)` is equivalent to `atomic_add(1, v)`, and `atomic_dec(v)` is equivalent to `atomic_sub(1, v)`. Various functions also return values based on the result, such as `atomic_add_return(...)`, which atomically returns the result of the addition. More functions can be discovered within `#include <asm/bitops.h>`.



- While contending, "sit and spin"
- Useful in *interrupt context*
- `#include <linux/spinlock.h>`
- `spin_lock()`, `spin_unlock()`
- `spin_trylock()`, `spin_is_locked()`

```
typedef struct { ... } spinlock_t

void spin_lock_init(spinlock_t *lock)

void spin_lock(spinlock_t *lock)

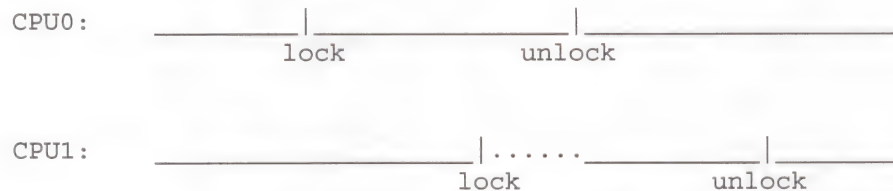
void spin_unlock(spinlock_t *lock)
```

Spinlocks implement a *optimistic* mutex, in that if a CPU contends on a spinlock, it assumes it will not be blocking long. While blocking, the spinlock performs a tight busy loop. Occasionally, it breaks from the loop to observe if the spinlock has been released (by some other CPU), in which case it breaks from the loop to acquire the lock.

Note that `spin_lock()` has no return value, as there is only one way the function will return: the CPU owns the lock.

Using Spinlocks

Spinlocks are used to synchronize two distinct CPUs:



Spinlocks are used to synchronize two distinct CPUs. While one CPU is holding a spinlock, one more more other CPUs might be pinned and performing no useful work, so a spinlock should be held for the minimum time necessary.

```
int spin_trylock(spinlock_t *lock)

int spin_is_locked(spinlock_t *lock)
```

As the names imply, `spin_trylock(...)` will try to acquire a spinlock, but immediately return with failure if the lock is not available, while `spin_is_locked(...)` will simply report the state of a spinlock.

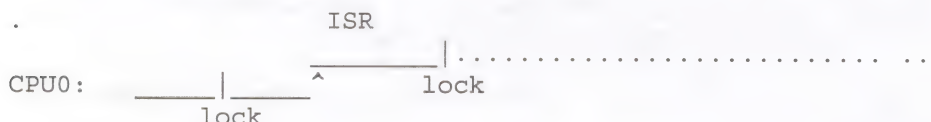


- Use case: Interrupt Service Routines (ISRs)
 - Spinlocks are non-blocking
- Disable local interrupts
 - When used outside of ISR
- `spinlock_irqsave()`, `spinlock_irqrestore()`

Spinlocks and local interrupts

Because spinlocks do not block (they spin instead), they are often used to protect data structures manipulated by *interrupt service routines*. When using the same data structure outside of an ISR, however, care must be taken to synchronize against local interrupts.

Consider the following, single CPU scenario. A CPU acquires a spinlock, then jumps to an ISR. The ISR then tries to acquire the same spinlock. Finding the spinlock already held, the ISR will spin until the lock is released. Deadlock.



```
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)
```

```
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)
```

The `spin_lock_irqsave(...)` function acquires the spinlock, determines (and records via the `flags` parameter) if interrupts were already disabled, and disables them if necessary. The complementary `spin_unlock_irqrestore(...)` function releases the spinlock, and re-enables interrupts if they were disabled by `spin_lock_irqsave(...)`. A programmer must allocate storage for the `flags` parameter, but does not manipulate the parameter directly.

```
spinlock_t foo_lock; unsigned long flags;
```

```
spinlock_irqsave(foo_lock, flags):
```

```
    /* critical section work occurs here... */
```

```
spinunlock_irqrestore(foo_lock, flags):
```

(Experienced C programmers may feel uncomfortable, because the API appears to pass `flags` by value instead of by reference, limiting its usefulness. These functions are actually macros, which accommodate accordingly.)



- Acquired as a "reader" or "writer"
- Support concurrent "readers"
- `typedef struct {...} rwlock_t`
- `read_lock(...), read_unlock(...)`
- `write_lock(...), write_unlock(...)`
- variants to suspend local ISRs

Read/Write Locks

A common variant of a mutex is a *read/write lock*, which can be acquired either as a reader or a writer.

Lock is:	Free	1+ Readers hold	Writer holds
Acquire as Reader?	Y	Y	N
Acquire as Writer?	Y	N	N

A writer precludes a reader or another writer, while a reader precludes only a writer. If multiple code paths are merely reading a data structure, not modifying ("breaking") it, they may do so concurrently.

Read/Write spinlocks

The Linux kernel provides a read/write variant of a spinlock as a `rwlock_t`. The read/write variant has a near identical API as a normal spinlock, simply replacing `spin` with `read` or `write`, as appropriate.



- Contending code paths "block"
 - yield the CPU via `schedule()`
 - only appropriate in *process context*
- `#include <linux/mutex.h>`
- `struct mutex`
- `mutex_lock(...), mutex_lock_interruptible(...)`
- `mutex_unlock(...)`

Mutexes

Until this point, this text has been referring to *mutexes* as a general term for mutual exclusion devices. Recent kernels now provide a specific structure also referred to as a `mutex`, which replaces many uses of `semaphores` found in previous kernels

Mutexes can be considered pessimistic locks. A contending code path is pessimistic that the mutex will soon be available, and yields the CPU by `scheduling()` off ("blocking"), allowing another process to schedule on. As an obvious consequence, mutexes are only appropriate in process context.

```
void mutex_init(struct mutex *mutex) DEFINE_MUTEX(mutexname)
```

Dynamically allocated mutexes should be initialized with `mutex_init(...)`, or the `DEFINE_MUTEX` declaration macro can be used to both declare and initialize a mutex. (The more conventional name `DECLARE_MUTEX` was already used, unconventionally, by `semaphores`).

```
void mutex_lock(struct mutex *lock)
```

```
int mutex_lock_interruptible(struct mutex *lock)
```

```
int mutex_trylock(struct mutex *lock)
```

```
void mutex_unlock(struct mutex *lock)
```

Mutexes are acquired with one of two variants of `mutex_lock...(...)`, which are potentially blocking calls. The two locking variants correlate with the two sleeping scheduling states of a process, `TASK_INTERRUPTIBLE` and `TASK_UNINTERRUPTIBLE`. With `mutex_lock(...)`, the sleeping task is not responsive to signals, and will only return once the mutex has been acquired. In contrast, a sleeping task is responsive to signals, with its return value qualifying if the mutex was acquired or if the task was awoken because it received a signal.

However the mutex is acquired, it should be released with `mutex_unlock(...)`, a non-blocking call.



- Mutexes appeared in 2.6.16 kernel
- Previous implementations used semaphores
- `#include <asm/semaphore.h>`
 - `struct semaphore`
 - `down(...), down_interruptible(...)`
 - `up(...)`
- New code should use Mutexes instead

Semaphores

Mutexes are a relative newcomer within the kernel. Where recent kernels use *mutexes*, older versions of the kernel used *semaphores*.

Semaphores are more general than mutexes, but in the vast majority of cases, semaphores were used as mutexes. Examination of the comments and structure definitions towards the top of `#include <linux/mutex.h>` provide motivation for the new mutex infrastructure, namely better debugging.



- "Real Time Clock" Management
- `#include <linux/rtc.h>`
- `struct rtc_device`
 - `struct mutex ops_lock`
 - `spinlock_t irq_task_lock`
- `drivers/rtc/interface.c`

Synchronization Example

The Real Time Clock infrastructure provides readable examples of using mutexes to protect structures which are only relevant in *process context*, and spinlocks to protect structures which are relevant in *system context*.



- Locks are expensive
- Sequential Locks
- *Read-Copy-Update (RCU)* algorithms
- Per-CPU design

Locking is Expensive

The overhead associated with the previously discussed synchronization structures is significant, because of both the mechanics of manipulating the structures, and the potential for contention.

More and more the Linux kernel is adopting designs which avoid acquiring global locks, including *sequential locks*, *Read-Copy-Update* algorithms, and *Per-CPU* design.



- Requirements
 - data is read often, updated seldom
 - multiple reads don't incur side effects
 - updates won't confuse readers
- `#include <linux/seqlock.h>`
- `write_seqlock(...), write_sequnlock(...)`
- `read_seqbegin(...), read_seqretry(...)`
- avoids `rwlock_t` writer starvation

Sequential Locks

A *sequential lock* is a clever approach to synchronization in which writers use a conventional spinlock, and readers avoid locking altogether (at the cost of possibly having to repeat an inconsistent read).

Sequential locks bind a traditional spinlock to a sequence number:

```
typedef struct {
    unsigned sequence;
    spinlock_t lock;
} seqlock_t;
```

```
void seqlock_init(seqlock_t *s)
```

```
DEFINE_SEQLOCK(name)
```

Dynamically allocated sequential locks are initialized with `seqlock_init(...)`, or sequential locks can be declared and initialized with the `DEFINE_SEQLOCK` declaration macro.

```
void write_seqlock(seqlock_t *sl)
```

```
void write_sequnlock(seqlock_t *sl)
```

Writers use these functions to acquire and release the sequential lock's spinlock, where the act of acquiring and releasing the spinlock each increases the associated sequence number by one. In other words, a complete write operation leaves the sequence number incremented by 2.

```
unsigned read_seqbegin(const seqlock_t *sl)
```

```
int read_seqretry(const seqlock_t *sl, unsigned iv)
```

Readers use the following approach:

```
do {
    seq = read_seqbegin(&foo_seq_lock);
    /* read the relevant data here ... */
} while (read_seqretry(&foo_seq_lock, seq));
```

Readers start by recording a copy of the sequence number, then perform a read. Readers must then confirm consistency by reexamining the serial number. `read_seqretry(...)` returns true under two conditions:

1. The initially acquired copy of the sequence number no longer matches the current sequence number.
2. The initially acquired copy of the sequence number is odd.

Either condition implies that a writer was modifying the data during the read, and the reader should try again. For the contested reader, the redundant reads are no worse than "spinning" the CPU, while for the un-contested reader, the spinlock is cleverly avoided altogether.



- Separates update from reclamation
- Requirements
 - Data is referenced via single pointer
 - Slightly stale data acceptable

Read-Copy-Update (RCU)

In certain use cases, the RCU mechanism can separate the act of updating from the act of reclamation, allowing both readers and writers to avoid locks altogether. The RCU mechanism anticipates dynamically allocated data referenced via a pointer, such as a linked list of structures.

A writer does not modify the data in place, but instead allocates a new element which it initializes with the updated data. Once initialized, the old element is replaced with the new element using an atomic pointer update. New readers will see the newly updated data.

Adding to the complexity, however, is the fact that existing readers could still be referencing the old copy of the data. The old element must somehow be tracked, and readers must somehow notify the RCU infrastructure that their read is complete, so the old data can eventually be reclaimed.

A more detailed discussion of the RCU mechanism is contained in the `Documentation/RCU` directory.



- Readers
 - `rcu_read_lock()`, `rcu_read_unlock()`
 - `rcu_dereference(p)`
 - No blocking allowed
- Writers
 - `rcu_assign_pointer(...)`
 - `call_rcu(...)`
- Reclamation
 - `synchronize_rcu()`

```
void rcu_read_lock()
```

```
void rcu_read_unlock()
```

```
void* rcu_dereference(void* p)
```

Readers of RCU protected data must notify the RCU infrastructure when they enter and leave reader critical sections using the `rcu_read_lock()` and `rcu_read_unlock()` global functions. Although the names are designed to be familiar to users of traditional locks, no locks are involved. By convention, readers are not allowed to block during critical sections, and preemption is disabled.

Readers should denote the dereferencing of a RCU relevant pointer with `rcu_dereference`. On some architectures, this forces relevant cache flushes, but also serves to document the process.

```
void* rcu_assign_pointer(void* ptr, void* new)
```

```
void call_rcu(struct rcu_head *head, void (*func)(struct rcu_head *head))
```

Writers update (publish) new data by updating the relevant pointers with `rcu_assign_pointer(...)`, which inserts appropriate barriers to prevent reordering of the initialization and publication, and also serves to document the process.

Writers submit a callback function to reclaim the old data using `call_rcu`.

```
void synchronize_rcu()
```

After a sufficient grace period, the reclamation infrastructure will call `synchronize_rcu()` to insure that all reader critical sections have completed, then execute the registered callback functions to reclaim stale structures.



- CPU specific structures avoid global locks
- Must still synchronize with ISRs
- Preemption must be disabled
- `#include <linux/percpu.h>`
- `DEFINE_PER_CPU(...)`
- `get_cpu_var(...), put_cpu_var(...)`

Per-CPU Design

More and more, the Linux kernel is adopting a design of "per-CPU" structures which avoid global locks altogether. Conceptually, the idea is simple. For any given data structure, imagine instead an array of structures, and if a code path is executing on CPU 2, it would use array element 2. Because no other CPU should be using array element 2, and no other code path can be executing on CPU 2 other than the current one, no locks are needed.

There are a couple of subtleties, however. First and foremost, kernel preemption must be disabled. If preempted, two different code paths on the same CPU could try to access the same structure "simultaneously", or more subtly, a code path on CPU 2 which gained a reference to element 2 might resume on CPU 4 instead, now using the wrong structure. Likewise, the kernel needs to accommodate "hot added" or "hot removed" CPUs.

Per-CPU API

```
DEFINE_PER_CPU(type, name)
```

```
per_cpu(var, cpu)
```

```
get_per_cpu(var)
```

```
put_per_cpu(var)
```

Most commonly, per-CPU structures are statically allocated and declared with the `DEFINE_PER_CPU` macro, which declares a collection of types as the symbol name:

```
DEFINE_PER_CPU(struct rq, runqueues);
```

The declared symbol then acts like a "handle" to the collection of values. The `get_per_cpu(var)` macro returns a lvalue for the element appropriate for the current CPU:

```
struct rq *this_rq;  
this_rq = get_cpu_var(runqueues);
```

For the reasons mentioned above, `get_cpu_var(...)` also disables preemption. When finished with the reference, a code path should "put it back", which re-enables preemption:

```
put_cpu_var(this_rq);
```

The `per_cpu(...)` macro can be used to obtain a reference to an element for an arbitrary CPU, and does not disable preemption.



- Event notification between threads
- `#include <linux/completion.h>`
- `struct completion`
- `wait_for_completion(...)` and friends
- `complete(...)`

Event Notification

Until now, we have been discussing synchronization as if the only concern was protecting data structures. Another concern is *event notification*, so that one kernel thread can block until an action has been performed by another thread.

```
struct completion { unsigned int done; wait_queue_head_t wait; }

DECLARE_COMPLETION(name)

INIT_COMPLETION(struct completion *c)
```

The Linux kernel uses *completions* for event notification. Two threads coordinating on an event share a `struct completion`, which is either statically declared with `DECLARE_COMPLETION(...)`, or dynamically allocated and initialized with `INIT_COMPLETION`.

```
void wait_for_completion(struct completion *)

int wait_for_completion_interruptible(struct completion *x)
```

The waiting thread invokes `wait_for_completion(...)`, or a variant thereof, to block until the event is complete. The return value of `wait_for_completion_interruptible(...)` is 0 if the event occurred, or `-ERESTARTSYS` if the thread was awoken by a received signal.

```
void complete(struct completion *)
```

The thread performing the awaited work invokes `complete(...)` to notify any waiting threads of the completion.



- Large coarse grained lock
- Difficult to exterminate
- Implemented as semaphore with differences
 - Supports recursive locking
 - Can "sleep" while held
 - Does not disable preemption
- `lock_kernel()`, `unlock_kernel()`

Retooling Code for Re-entrancy

When approaching a body of non-reentrant code with the goal of making it re-entrant safe, a common strategy is to begin with a small number of large, coarse grained locks which protect multiple systems. As more effort is expended, individual subsystems are protected with more fine-grained locks. Eventually, each individual structures are protected with dedicated locks.

The difficult part of this process is deciding when structures are suitably protected with fine-grained locks, and the the coarse grained overarching lock can be removed.

Linux 2.0

Developers decided that the 2.0 version of the Linux kernel should gain SMP support. To achieve this goal, one global spinlock was created. The lock was acquired upon entering the kernel, and released upon resuming userspace.

This allowed Linux to be an SMP system, but in a way that limited scalability.

Linux 2.6

Obviously, there has been much work done to increase the performance since the Linux 2.0 (but SMP capable!) kernel, yet the original lock is still present, affectionately known as the BKL, or "Big Kernel Lock". Removing instances where the lock must be held is tedious work, and for seldom used parts of the kernel (such as initialization), not yet worth the effort.

Mercifully, the lock is more forgiving than the original spinlock.

1. The BKL can be acquired recursively.
2. A task can sleep while holding the BKL.
3. The BKL is now implemented as a semaphore, not a spinlock.



End of Lecture 6

- Questions and Answers
- Summary
 - Atomic bit and integer operations
 - Spinlocks
 - Mutexes
 - Completions
 - Read-Copy-Update (RCU)
 - Big Kernel Lock



Lab 6.1: Safely Querying Process Task Structures

Deliverable: Updated kernel modules which safely accesses the list of current tasks running on the system

Instructions:

1. In the last lab, kernel API functions were used to access and create linked lists of data structures. Both exercises queried the list of task structures for all of the processes running on the system. What would happen if the list of processes changed while your program was running? This is where synchronization methods come in.

Update your kernel modules from the previous lab to safely access the list of task structures for the system. You will have to do a little research using **cscope** or **LXR** to figure out how to properly lock the task list so you can access it safely.

Since the linked list of `pinfo` structures is private to your kernel module and the kernel won't allow you to load your module multiple times, you don't need to protect your linked list with a synchronization method at this point.

Lab 6.1 Solutions

1. One way to figure out which locking methods and locks are available for the task structures of current processes is to look at existing code. Use **cscope** or **LXR** to look at other places in the kernel where the task list is browsed. Search for instances where the `for_each_process()` macro is used for examples of locking.

The solution for this exercise can be found on the instructor machine at `/var/ftp/pub/rhd361-solutions/06-synchro`. The source code for the solutions which use synchronization methods are called `listtasks.c` and `linkedlist.c`.





Kernel Debugging 1: Tools and Techniques

Upon completion of this unit, you should be able to:

- Describe kernel debug tools and interfaces
- Configure target system for kernel **crash** analysis
- Describe **crash** tool operations and requirements
- Configure target system for **crash** analysis
- Invoke **crash** against a variety of targets
- Perform live system analysis with **crash**
- Perform batch analysis with **crash**



- Default kernel image cannot be used for debugging
 - Compressed
 - No debugging symbols available
- Debuggers require an uncompressed object file that includes debugging symbols
 - Generated at build time as `vmlinux`
- Provided by two packages
 - `kernel-debuginfo-common` : provides files common to all kernel builds
 - `kernel-debuginfo` : provides the binary object files for one specific build (such as the Xen, PAE, or regular kernel)
- Packages are available online:
 - `# yum --enablerepo=rhel-debuginfo install kernel-debuginfo`
 - `ftp.redhat.com:/pub/redhat/linux/enterprise/`
 - `http://people.redhat.com/anderson/debuginfo.html`

Debuggers require object files that contain debugging symbols in order to load and examine executable files. The kernel is no exception. For performance and efficiency reasons, Red Hat by default only ships a compressed kernel that was stripped out of debugging symbols, available under `/boot/vmlinuz-kernel_version`. Such a kernel is obviously useless for debugging purposes.

To debug a kernel, one needs a very special file installed on the system: `vmlinux`. This file is usually generated at compile time, and can be configured to contain debugging symbols needed for further analysis. `vmlinux` is not compressed, so its size is generally about ten times the size of the default kernel available under `/boot`.

Two packages are publicly available that contain `vmlinux`: `kernel-debuginfo-common` and `kernel-debuginfo`. The `kernel-debuginfo-common` package contains files that are common to all builds for a specific version. The `kernel-debuginfo` package, on the other hand, contains the files that are specific to one particular build, such as the Xen, PAE, or regular kernel.

The `kernel-debuginfo` package can be installed from RHN by enabling the `rhel-debuginfo` repository:

```
# yum --enablerepo=rhel-debuginfo install kernel-debuginfo
```

or be downloaded from Red Hat's anonymous FTP site:

```
ftp://ftp.redhat.com:/pub/redhat/linux/enterprise/
```

The following web page offers a helpful interface for finding the the right `kernel-debuginfo` package on the FTP site: `http://people.redhat.com/anderson/debuginfo.html`



- Debugger requires that `vmlinux` and running kernel be generated from the same build version
- `kernel-debuginfo` must match:
 - kernel version
 - kernel variant (like Xen or PAE)
 - kernel architecture
- RPM-packaged files are installed under:

```
/usr/lib/debug/usr/src/kernels/kernel-version/
```

- Debuginfo packages require around 600MB of disk space once installed
- Different "flavors" of the same-versioned debuginfo package install to the same directory!

A common mistake is to install debuginfo packages that come from a kernel build that is different from the kernel you are trying to analyze. The debuginfo package installed must match the kernel version, variant (like Xen or PAE) and architecture. Make sure you obtain the right package version by taking a look at the output of the **uname** command:

```
$ uname -r
2.4.21-38.ELsmp
```

The debuginfo packages install files under `/usr/lib/debug/usr/src/kernels/$(uname -r)/`. These files can require as much as 600MB of disk space, so make sure you have plenty of space available before you install them.

Be aware that each of the different "flavors" of the debuginfo package install their source code to the same directory, so the directory will hold the source code for the *last* debuginfo package that was installed.



- CPUs operate in two different environments:
 - User space
 - Kernel space
- User space provides a protected environment for applications:
 - Independent memory space for each process
 - Time available on the CPU determined by the scheduler
 - Preemptible
 - Programming errors only terminate the process
- Kernel space allows unlimited access to memory and hardware:
 - No safety net - an error can make the entire system crash or hang
 - One unique memory space for the kernel, drivers and all kernel threads
 - Non-preemptible, with a few exceptions
 - Functions can run in the kernel on behalf of processes

Code executing on the CPU may be invoked in user or kernel space. This has several important effects:

- User space code runs on top of a "safety net" that isolates badly programmed applications from the rest of the system. That way, bugs in an application will not affect other processes or the kernel.
- Applications running in user space have their own private address space. Since it is independent from other processes, this address space cannot be used, by default, for inter-process communication.
- Code running in kernel space has access to everything. All addresses or hardware devices may be directly accessed.
- Likewise, there is no limit to the amount of damage that can be done in kernel space. While errors are limited to user space in the case of an application, kernel code can modify virtually everything.
- An unrecoverable error in kernel space will bring the entire system down and cause a kernel panic.



- Debugging can be done live on the running kernel:
 - Limited to read operations
 - No way to single-step into the code
 - Essentially used to monitor the current state of the kernel
- Postmortem debugging involves analyzing a kernel crash:
 - Kernel is not running anymore
 - Dump of the memory content was obtained at crash time
 - We need to know how and where the kernel crashed

Live debugging involves having a debugger, **gdb**, examine the memory which is currently in use by the kernel. As opposed to regular applications, debugging a kernel live involves a few restrictions:

- No write operations since modifying the memory interactively could make the entire system crash.
- No way to suspend kernel execution since the debugger runs on top of it.
- Similarly, single-stepping through the code is not possible.

Postmortem analysis depends on the ability to dump the system memory to a file right before a crash (this is what we call the crash dump) and examine it latter on. We usually use this method to determine what part of the kernel code caused the crash so as to find out if the problem was caused by a bug.



- Kernel hangs and crashes are fundamentally different:
 - Hangs in kernel space are often temporary and caused by time-consuming operations, and no oops message is displayed
 - Crashes are usually instantaneous and lead to a kernel oops message, and a reboot must follow
- Diagnosing a kernel crash involves obtaining the kernel crash dump from memory
- Kernel hangs are more complicated: we need to determine what the kernel is executing
- In all cases the entire system is affected
- An application hang or crash only affects the process that runs the application

Events that affect an entire Linux system are usually separated in two different classes: crashes and hangs. Both need to be investigated in order to determine the root cause but they require different tools and strategies.

Kernel crashes often immediately follow a problem in kernel space. A programming error, a defective piece of hardware, an unsupported operation can all lead to a kernel crash. In these cases, an oops message is displayed on the console to help with diagnosis and the entire system is interrupted.

Kernel hangs are more subtle: they can be as simple as temporary performance problems caused by inefficient algorithms or as complicated as dead locks that freeze the system forever. No kernel oops message is displayed on the console, which can make the troubleshooting process tedious. Tools such as the SysRq key may be helpful here in order to trigger an event that will assist us with debugging. Watchdogs can be used here to trigger a crash if the kernel is hung for a specific period of time.

Applications rarely make the entire system crash or hang. Events in user space are isolated in such a way that the system and other processes will not be affected. System calls invoked by applications may, however, trigger some operations in kernel space that will eventually cause a crash or a hang.



- Device drivers shipped by Red Hat can be analyzed just like regular kernel code from the kernel image
- Proprietary drivers make debugging more complicated:
 - Object code needs to be available
 - The code has to include symbols
- Tainted flag indicates if a proprietary driver was loaded in the past on a running kernel
 - Indicates if a proprietary driver was loaded in the past
 - State is available under `/proc/sys/kernel/tainted`
 - State is also included in kernel crash message

We will see in a latter chapter that kernel modules, which typically implement device drivers, can be debugged just like code from the kernel image. The only requirement, as always, is to make sure the `kernel-debuginfo` package is installed.

Proprietary kernel modules can be a bit trickier since the source code may not necessarily be available. In order to properly debug a kernel module, the manufacturer has to provide object files that we compiled with debugging symbols.

Another thing to note about proprietary modules (or any non-GPL module for that purpose) is that they will taint the kernel. Kernel tainting involves a flag, the taint flag, being switched on whenever a proprietary module is loaded. This is useful since it lets supports engineers determine if a module outside of Red Hat's kernel source tree was loaded in kernel space. This is a problem since Red Hat does not have access to the source code of such modules: Red Hat engineers cannot find bug and fix them.

The tainted flag is displayed in the kernel log messages whenever its state gets modified. Similarly, it will show up in the crash message and can also be monitored at any time by looking at `/proc/sys/kernel/tainted`.



- **dmesg**
 - Shows kernel messages since boot
 - `/var/log/dmesg`
- **strace**
 - Traces system calls invoked by an application in user space
 - Can profile all executables, including scripts
- **ltrace**
 - Can trace both library and system calls made by an application in user space
 - Can profile elf binaries only
- **pstack**
 - Shows the current user space stack of a process
- **pmap**
 - Displays the memory map of a process in user space

User space debugging tools, although they are not designed for kernel debugging, are often needed in order to gain more knowledge about what is running on top of the kernel.

The **dmesg** is useful to determine if any kernel message got printed due to an error. Issues that caused kernel messages may be followed by kernel panics, so it may be wise to keep an eye on what gets logged. **dmesg** outputs the contents of a ring buffer, whose earliest contents may have been lost. For a complete record, view the file `/var/log/dmesg`.

Kernel issues, while almost always caused by code in the kernel, may be triggered by processes running in user space. Monitoring these processes involves looking at which *system calls* get invoked. The **strace** can easily start a process or attach to an existing process and list all system calls executed.

ltrace is almost identical to **strace** but concentrates on library calls.

Looking at the state of a process may require an understanding of the state of its stack. The **pstack** command will let you take a look at all functions that have not yet returned in the context of a process. **pstack** can also list stacks of all threads currently running in a process.

The **pmap** command lets us monitor the memory space of a process. This includes all libraries used, memory mapped files and allocated memory areas.



- /proc provides information about the current state of the kernel
- Of particular interest for debugging purposes:
 - /proc/PID#
 - /proc/sys/
 - /proc/kcore
 - /proc/kallsyms
- Data under /proc is generated on the fly and is not persistent

The /proc filesystem is used to expose kernel data that would otherwise be unavailable in user space. This includes process information, hardware device data, various statistics and counters as well as many tunable settings used for performance optimization.

Kernel debugging sometimes involves gathering information about the system state and what is running. This can be accomplished by looking at specific files and directories under /proc.

Process information has historically been made available under /proc/PID#. Each process currently running on the system has its own directory under which a number of attributes such as open files, memory consumption, memory map, process statistics, scheduling data, etc. can be obtained. User space utilities such as **ps**, **top** and **mpmap** rely heavily on this section of the /proc filesystem.

Kernel tunable settings are generally provided under /proc/sys/. These settings sometimes can explain why the kernel is behaving in certain ways. Moreover, the root user may modify the behaviors by modifying files under this directory.

Other files are available for read purposes under /proc. /proc/kcore, for instance, provides a way to access the system physical address space. /proc/kallsyms can be useful to list all function and variable names exported in the kernel. We will use some of them later in this course when the **crash** tool will be covered.

Always keep in mind that entries under /proc keep changing all the time since this is a dynamic filesystem (i.e. data is not stored on disk and file content is generated at read time). Likewise, none of the files under /proc are persistent; rebooting the system will reinitialize all files found under this filesystem.



- `/proc/sys/kernel/panic`
- Determines what to do after a kernel crash
 - A value of 0 will wait for the user to reboot the system
 - A positive value represents the number of seconds to wait before rebooting
- Can be set persistently using the `kernel.panic sysctl` setting

By default a kernel crash will cause an oops message in the kernel log, and the kernel will panic right after. This causes the system to freeze, waiting for the user to manually reboot it. This can be a bad thing if we do not have physical access to the system or if we are highly concerned about uptime.

A solution to this problem is to force the kernel to reboot automatically after a specified number of seconds. This configuration is available under `/proc/sys/kernel/panic` and is set to 0 (disabled) by default. To enable it, just write the delay (in seconds) to the file:

```
# echo 10 > /proc/sys/kernel/panic
```

Alternatively, one can also make the setting persistent by adding the following entry to `/etc/sysctl.conf`:

```
kernel.panic = 10
```

Make the change take effect immediately and verify:

```
# sysctl -p
# sysctl kernel.panic
kernel.panic = 10
```

The Linux kernel also supports a boot argument called **panic** that can be passed from the bootloader. Adding the following argument to the **kernel** line in `/boot/grub/grub.conf` will do the trick:

```
panic=10
```



- Mostly used to monitor device drivers and kernel subsystems
- Contains hardware information not available under `/proc`
- Not persistent and may change based on system state

The 2.6 kernel brought several enhancements for device drivers, including a new model that is used to represent hardware device in a uniform way across the entire system. In practice, the Linux Device Model constitutes a way to monitor standard pieces of data, such as the interrupt number, in the same way for all devices. The `/sys` filesystem reuses this model to bring the data it maintains to user space.

`/sys` behaves just like `/proc`: files can be read and, sometimes, modified. `/sys` is also dynamic: plugging in a new hardware device may cause a new directory or file to be created.

`/sys` is not used directly by kernel debuggers but can become useful when more information is required for troubleshooting purposes.



- Optional filesystem, not mounted by default
- Exposes kernel variables to user space
- Values can be read and modified
- Can be activated with:

```
# mount -t debugfs none /sys/kernel/debug/
```

- Content depends on which drivers are currently loaded

The debug filesystem is a new feature shipped under Red Hat Enterprise Linux 5. Although it is not enabled by default, it can easily be made available with the **mount** command:

```
# mount -t debugfs none /sys/kernel/debug/
```

Some device drivers and kernel subsystems use debugfs to expose information that would otherwise be hard to provide through `/proc` or `/sys`.



- Kernel developers often use printing as a debugging tool
- Printed data is available through:
 - **dmesg**
 - `/var/log/messages` (default)
 - `/proc/kmsg`
- Appropriate for small, infrequent log entries
- Does not work well for large quantities of data

Printing constitutes the most basic, often the easiest, method to troubleshoot problems in the kernel. Through the use of the **printk()** function, kernel developers can print any variable value, message or data they wish. While this is very flexible, it requires kernel programming knowledge and recompiling the kernel (or a specific module).

Messages from **printk()** can be monitored with the **dmesg** command, which reads the messages directly from a kernel ring buffer that holds them.

Kernel messages are also sent to the file `/proc/kmsg`. If `klogd` is running (it is started as part of the `syslog` service), it consumes the `/proc/kmsg` messages and passes them on to `syslogd`. Note: only one reader is allowed at a time on `/proc/kmsg`.

The `/var/log/messages` file is populated with these kernel messages by default because of the following line in `/etc/syslog.conf`:

The `syslogd` daemon's configuration file, `/etc/syslog.conf`, determines how facility messages are handled. By default, all kernel messages are appended to the file `/var/log/messages` because of the line:

```
*.info;mail.none;authpriv.none;cron.none          /var/log/messages
```

Kernel messages can be redirected elsewhere by uncommenting the line:

```
kern.*          /dev/console
```

and altering the destination.

Any changes to `/etc/syslog.conf` require reloading the `syslogd` daemon to force a re-read of its configuration file:

```
# service syslog reload
```

While **printk()** is a valid way to catch error conditions that occur in kernel code, printing is not considered a valid technique for advanced debugging that requires looking at large quantities of data.



- Message that reports a bug or a hardware problem
- Not always fatal - the system may keep running
- Always printed to the console before a kernel panic (i.e. a crash)
- Contain useful information for basic debugging

When an English-speaking person has a minor accident or a stumble, the sound “oops!” is often heard. The Linux kernel also stumbles at times and also produces an oops report. An oops message is written into the kernel message buffer, and subsequently migrated to the syslog daemon for output, when the kernel encounters a machine fault such as a NULL pointer dereference. The execution context for the kernel thread encountering the fault is written to the kernel log. The faulting instruction, the CPU register contents, and a limited amount of backtrace are all recorded in the logs.

In some situations a kernel Oops will be printed but the kernel will keep running normally. This is usually caused by an event in the kernel that required some attention. In some other cases the Oops message will be printed and the kernel will panic (or crash) shortly after. The Oops can therefore become extremely helpful in order to know what happened right before the kernel panic.

Information included in the Oops message includes:

- Text description of the problem encountered
- CPU the Oops occurred on
- CPU registers content
- Tainted flag value
- Kernel version
- Stack backtrace
- Code executing on the CPU
- Call trace

There used to be a tool called **ksymoops** provided to convert hex addresses to real function names in Oops messages. This is now useless in 2.6 kernels since the code that generates Oops messages is now smart enough to display names as well as addresses.

The <http://www.kerneloops.org> website collects kernel oops and warning reports from various mailing lists and bugzilla tickets.



- Enable the SysRq feature:
 - `# echo 1 > /proc/sys/kernel/sysrq`
 - To persist reboots add the following line to `/etc/sysctl.conf`:
 - `kernel.sysrq = 1`
- Keystroke combination (from virtual console):
 - `Alt-SysRq-c`
- Keystroke combination (from within X):
 - `Ctl-Alt-SysRq-c`
 - Traceback and dump results are sent to `/var/log/messages`
- Non-interactive:
 - `# echo c > /proc/sysrq-trigger`
 - SysRq does not need to be enabled first

A system may become unresponsive due to memory contention, an overloaded CPU, high priority tasks taking over the entire system, etc. Those situations may justify the need for a mechanism that lets you force specific commands in order to put the system back into a specific state. Also, a kernel developer or system administrator may want to test the configuration of `kdump/kexec` by simulating a kernel crash before putting the machine into production.

The Linux Magic System Request Key (SysRq) kernel feature is used specifically for these purposes. SysRq is disabled by default on Red Hat Enterprise Linux 5 kernels. Once enabled (`echo 1 > /proc/sys/kernel/sysrq`), typing the keystroke combination `Alt-SysRq-c` from a virtual console will cause an immediate crash of the machine. When running `X`, the window manager might capture the `Alt` keystroke combination, so it must be preceded with a `Ctl` keystroke (e.g. `Ctl-Alt-SysRq-c`).

In order for this feature to persist a reboot, configure this parameter in `/etc/sysctl.conf`:

```
kernel.sysrq=1
```

Note: the command `echo c > /proc/sysrq-trigger` does not require that the SysRq feature be enabled first.

An important subset of the single-letter SysRq commands is listed here:

c	Force the system to crash and generate a crash dump
t	Display all current tasks and their information
m	Display memory information
i	Send SIGKILL to all processes including <code>init</code>
e	Send SIGTERM to all processes except <code>init</code>

For more information about SysRq and its options, refer to `Documentation/sysrq.txt` in the kernel source tree.



- The **sosreport** command gathers data about a system's hardware and configuration
- Includes RPM packages, `/proc` and `/etc` data
- Requires root access
- Usually takes a few minutes to generate and can slow down the system
- Creates archive under `/tmp/sosreport` (<1MB disk space)
- Usually is the first thing to provide Red Hat for support

Support organizations often need to repeatedly ask clients for the same pieces of information following a crash: system and kernel version, architecture, hardware devices, configuration files, etc. The **sosreport** automates this task by generating an archive that contains everything Red Hat support engineers typically require in order to analyze a problem.

sosreport scans a number of files and directories, including `/etc`, `/proc` and the RPM database, and stores all the information in an archive under `/tmp`. **sosreport** needs to be executed as root due to the nature of the data it gathers on the system.

The resulting file can then be transmitted to Red Hat or other people involved in the troubleshooting of an issue.

Note: **sosreport** replaced the previously-used **sysreport** in RHEL 5.0.



- **crash** is a kernel-aware analysis utility implementing a superset of gdb capabilities
- Provides sophisticated symbolic kernel debug capability
- Handles a variety of targets, including live memory and vmcore files
- See Dave Anderson's whitepaper for complete details:
 - http://people.redhat.com/anderson/crash_whitepaper/

The Red Hat crash analysis utility is loosely based on the SVR4 UNIX **crash** command, but has been significantly enhanced by completely merging it with GNU's **gdb** debugger. The marriage of the two effectively combines the kernel-specific nature of the traditional UNIX crash utility with the source code level debugging capabilities of **gdb**. The **crash** utility can be used to investigate a wide variety of kernel core dumps, including:

- Live Linux systems
- Linux kernel core dumps created by the Kdump, `netdump`, and Diskdump facilities
- Compressed Linux kernel core dumps created by the **makedumpfile** command and Diskdump facility
- Xen host Linux kernel core dumps created by the Kdump facility
- Xen guest Linux kernel core dumps created by the original and ELF-format xendump facility
- Xen hypervisor core dumps created by the Kdump facility
- s390/s390x Linux kernel core dumps created by the IBM standalone core dump facility
- Linux kernel core dumps created by the LKCD (Linux Kernel Crash Dumps) Sourceforge project
- Linux kernel core dumps created by the Mcore patch offered by Mission Critical Linux

The current set of **crash** commands consist of common kernel core analysis tools such as kernel stack back traces of all processes, source code disassembly, formatted kernel structure and variable displays, virtual memory data, dumps of linked-lists, etc., along with several commands that delve deeper into specific kernel subsystems. Relevant **gdb** commands may also be entered, which in turn are passed on to the **gdb** module for execution.

The crash utility is designed to be independent of Linux version dependencies. When new kernel source code impacts the correct functionality of **crash** and its command set, the utility will be updated to recognize new kernel code changes while maintaining backwards compatibility with earlier releases. The most current version of the **crash** utility may be found here: <http://people.redhat.com/anderson>.



- **crash** requires a valid kernel memory image to interpret
 - Can be live or vmcore
 - Live system memory: `/dev/crash` (requires root privileges)
- **crash** requires a matching kernel object file to guide its interpretation of the memory image
 - Must include symbolic information
 - `kernel-debuginfo` package
 - `/usr/lib/debug/lib/modules/<kernel-version>/vmlinux`
- Must run **crash** on same architecture as memory image source
 - Example: To examine a PPC vmcore, run **crash** on a PPC machine

The **crash** utility requires a valid kernel memory image and matching kernel object file.

The memory image may consist of a kernel crash dump file generated from any of the supported dump facilities, or live system memory accessed via `/dev/mem` or its replacement in RHEL4/RHEL5, the `/dev/crash` driver. If no dump file argument is issued on the **crash** command line, live system memory will be used by default. When examining a live system, root privileges are required.

The `vmlinux` kernel object file must have been built with the `-g` option so that it will contain the debug data required for symbolic debugging. In RHEL4 and RHEL5 installations, the `vmlinux` kernel object file is part of the `kernel-debuginfo` package and is found in the `/usr/lib/debug/lib/modules/<kernel-version>` directory.

The crash utility is actively developed and tested on the x86, x86_64, ia64, ppc64, s390 and s390x processors. Legacy support for the Alpha and 32-bit PowerPC platforms exists, but no longer actively maintained.

The crash utility is backwards-compatible from at least Red Hat 6.0 (Linux version 2.2.5-15) through Red Hat Enterprise Linux 5 (Linux version 2.6.18+). Immediate support for the latest kernel versions cannot be guaranteed but modifications are constantly being implemented to support changes in upstream kernel versions. **crash**'s developmental goal is to keep the utility independent of Linux version dependencies and build in recognition of major kernel code changes and new kernel versions, while maintaining backwards compatibility.



- **crash** is included in the Development Tools package set
 - Select during or after system installation
 - `# yum groupinstall "Development Tools"`
- Otherwise, use **yum** or **rpm** as follows:
 - `# yum install crash`
 - `# rpm -ivh crash-version.arch.rpm`
- Obtaining the latest version is always a good idea

Starting with the RHEL3 release, the `crash` package is automatically installed during system installation if the Development Tools package set is selected.

A list of installed and available package sets can be viewed with the command:

```
# yum grouplist
```

The latest "upstream" version of the `crash` utility, available in RPM and gzip-compressed tar file formats, can be found at: <http://people.redhat.com/anderson>.



- Typical postmortem debugging:
 - `# crash vmlinux vmcore`
 - Kernel object file and memory image are supplied, respectively
- Live memory debugging:
 - `# crash vmlinux`
 - `/dev/crash` used by default for live memory image
- Live memory debugging (with `vmlinux search`):
 - `# crash`
 - Pre-defined directories are searched for proper `vmlinux`
 - Version string matched to the running kernel (`/proc/version`)

When `crash` is invoked, at least two arguments are always required:

1. The kernel object filename, often referred to as the kernel namelist. When initially built from the kernel sources, its name is `vmlinux`. In RHEL4 and RHEL5 installations, the `vmlinux` file is part of the `kernel-debuginfo` package, and located in `/usr/lib/debug/lib/modules/<kernel-version>/`.
2. The kernel memory image or dumpfile name, typically named `vmcore`.

When **crash** is run on a live RHEL4 or RHEL5 system, `/dev/crash` is used as the memory image. Previous versions of Red Hat Enterprise Linux used `/dev/mem`, which is now restricted on x86 and x86_64 systems.

Because **crash** can also search standard directories, if the `vmlinux` file is in any of the following standard locations, no arguments to the **crash** command are necessary for it to find and use it.

- `/`
- `/boot`
- any subdirectory of `/usr/src`
- `/usr/lib/debug/lib/modules/<kernel-version>`
- `/usr/src/redhat/BUILD/kernel-<kernel-version>/linux-<kernel-version>`

crash performs a search in all of the directories above until a kernel object file (`vmlinux-<kernel-version>`) is found with a version string matching the running kernel, as indicated by `/proc/version`. If a matching kernel is found, then `crash` may be invoked on a live system simply by entering:

```
# crash
```



```
KERNEL: ✓  
/usr/lib/debug/lib/modules/2.6.18-53.1.6.el5/vmlinux  
DUMPFILE: /dev/crash  
CPUS: 4  
DATE: Sun Feb 10 04:01:21 2008  
UPTIME: 12 days, 12:27:31  
LOAD AVERAGE: 0.00, 0.00, 0.00  
TASKS: 173  
NODENAME: host1.example.com  
RELEASE: 2.6.18-53.1.6.el5  
VERSION: #1 SMP Wed Jan 16 03:56:43 EST 2008  
MACHINE: i686 (1862 Mhz)  
MEMORY: 4 GB  
PID: 10714  
COMMAND: "crash"  
TASK: f702caa0 [THREAD_INFO: f3cfc000]  
CPU: 2  
STATE: TASK_RUNNING (ACTIVE)
```

Given that all invocation arguments are in order, you should see summary information similar to that shown above. This example shows a successful invocation using live memory on a running system, as indicated by the `/dev/crash` value for the dump file.

Invocation errors will cause the crash session to abort upon initialization. Typically they occur as the result of one of the following reasons:

1. The `vmlinux` file contains no debug data (i.e., was built without the `-g` flag), and no additional debug kernel object file name was entered on the command line
2. The `vmlinux` file does not match the dumpfile
3. The `vmlinux` file could not be found on a live system
4. The associated debuginfo file cannot be found
5. The machine type of the **crash** utility binary does not match the machine type of the `vmlinux` and/or `vmcore` arguments



- The **crash** tool has tremendous capabilities, so you will eventually need to consult the help facilities to tap its full power
- Offline help
 - # `man crash`
 - # `crash --help`
 - # `crash -h command`
- Interactive help
 - `crash> help command`

Type the following command to get an explanation of the **crash**'s command line arguments:

```
# crash -h
```

For a list of supported commands:

```
# crash -h commands
```

For details on a particular command (using the `waitq` command as an example):

```
# crash -h waitq
```



- **crash** initializations (in order of execution)
 1. `~/ .crashrc`
 2. `$PWD/.crashrc`
- Batch
 - `# crash -i commandfile`
 - `# crash < commandfile`
 - `# echo bt | crash -s vmcore vmlinux`
- Interactive
 - Command history and editing
- Shell commands with `!` prefix:
 - `crash> !df`

Upon a successful session invocation on a dump file or a live kernel, the `crash>` prompt will appear. Interactive crash commands are gathered using the GNU readline library, taking advantage of its command line history mechanism, and its **vi** or **emacs** command line editing modes. Commands may also be issued to crash from a file.

The command line history mechanism of **crash** consists of a numbered list of previously-run commands. The full list of commands may be viewed by entering `h` at any time. For example:

```
crash> h
[1] bt -a
[2] ps
...
```

The command line editing mode of **crash** may be set to either **vi** (the default) or **emacs**. The mode may be set via the `EDITOR` environment variable or by creating a `'set vi'` or `'set emacs'` entry in a `.crashrc` file in the user's `$HOME` directory:

Command line input can be fed into **crash** from a file in any of the following ways:

1. Upon invocation, as in:

```
# crash -i inputfile
```

2. Upon invocation, by entering the commands in a `.crashrc` file, which can be either in the user's `$HOME` directory or in the current directory.

3. On the command line during a crash session, as in:

```
crash> < inputfile
```



- Default radix for non-pointer values is decimal
- Verbose output paged via **less**
- Output can be piped and redirected

By default, command output that would overflow the user's display screen is piped to `/usr/bin/less`. This default output scrolling behavior can be turned off by entering `'set scroll off'` in a `.crashrc` file located in either the `$HOME` or current directories:

Command output may be redirected to a pipe or to a file using standard shell redirection syntax:

```
crash> task | grep uid
uid = 3369,
euid = 3369,
crash> foreach bt > bt.all
crash> ps >> process.data
crash> kmem -i | grep SLAB > slab.pages
```

The default numerical output radix for non-pointer values is decimal, which is most often noticed when using the built-in **gdb** capability of printing formatted data structures. During runtime, the `'set radix NN'` commands (or their respective built-in aliases) can be used to toggle the output radix:

```
crash> set radix 16
output radix: 16 (hex)
crash> dec
output radix: 10 (decimal)
crash> hex
output radix: 16 (hex)
```

Alternatively, the `px` or `pd` aliases coerce the "print" command `p`, to override the current output radix:

```
crash> p jiffies
jiffies = $4 = 69821055
crash> px jiffies
jiffies = $5 = 0x42963aa
```




- Four main categories:
 - Kernel symbolic display
 - System state commands
 - Utility functions
 - Session control commands

Each crash command generally falls into one of the following categories:

Symbolic display of kernel text or data: These commands typically take full advantage of the power of gdb to display kernel data structures symbolically:

struct	union	whatis	sym
dis	p	*	

System state: The majority of crash commands come from the following set of "kernel-aware" commands, which delve into various kernel subsystems on a system-wide or per-task basis. The task-specific commands are context-sensitive, meaning that they act upon the current context unless a PID or task address is specified as an argument:

bt [†]	dev	files [†]	fuser
irq	kmem	mach	mod
mount	net [†]	ps	pte
runq	sig [†]	swap	sys
task [†]	timer	vm [†]	vtop
waitq			

Utility functions: These are a set of useful helper commands serving various purposes, some simple, others quite powerful:

ascii	btob	eval	list
ptob	ptov	search	rd
wr			

Session Control Commands: These commands typically aid in the efficient running of a crash session:

alias	exit	extend	foreach
gdb	repeat	set	q
!			

[†]Indicates commands that are context-sensitive.



- On dumpfiles, the default context when **crash** starts will be the task that was running when:
 - `die()` was called
 - `panic()` was called
 - `Alt-SysRq-c` input at keyboard
 - `# echo c > /proc/sysrq-trigger`
- On live systems, the default context is the **crash** process itself
- Use `set` to change context

Upon a successful invocation of a **crash** session, one of the existing Linux tasks is selected as the current context. It is important to be aware of the current context because several **crash** commands are "context-sensitive", meaning that the command is executed from the view-point of the current context. Therefore, the output of context-sensitive commands can vary depending upon which context is current.

During runtime, the current context can always be displayed by entering the `set` command with no arguments:

```
crash> set
PID: 1696
COMMAND: "insmod"
TASK: c74de000
CPU: 0
STATE: TASK_RUNNING (PANIC)
```

The current context can be changed to a new task via the `set` command. Either of two "handles" may be used to identify a task, the PID number, or the kernel address of the task's `task_struct`. Alternatively, the current context can be set to the task running on a given CPU number using `'-c CPU-number'` or back to the panicking task using the `'-p'` option, as follows:

```
crash> set -p
PID: 1696
COMMAND: "insmod"
TASK: c74de000
CPU: 0
STATE: TASK_RUNNING (PANIC)
```



End of Lecture 7

- Questions and Answers
- Summary
 - User-space kernel analysis tools
 - **crash** for kernel debugging



Lab 7.1: Preparing a System for Kernel Debugging

Scenario: Your system will be used as a kernel debugging workstation. You need to install all the required components to compile kernel code and run the debugging utilities.

Instructions:

1. Make sure the `kernel-headers`, `kernel-doc` and `kernel-devel` packages are installed on your system. ✓
2. Install the `kernel-debuginfo-common` and `kernel-debuginfo` packages. Determine where the `vmlinux` file got installed.

~~/usr/src/redhat~~
/usr/lib/debug/lib/modules/2.6.../vmlinux

Lab 7.2: Gathering System Information

Scenario:

You have been asked to investigate a problem that shows up on a system that you have never used before. We will invoke the utilities covered earlier in this chapter to investigate how the system is configured and what it is running.

Instructions:

1. Switch to the text mode interface (*Ctrl-Alt-F1*) and log in as root.

Determine the architecture and the version of the kernel that is currently running on your system.

2.6.18 SMP 686 386

2. Install the `vsftpd` package and start the `vsftpd` daemon. Make sure it restarts automatically when the system reboots.

service vsftpd status ✓

Determine the process identification number (PID) of the `vsftpd` daemon you just started.

3. Display the memory map of the `vsftpd` daemon.

-ps aux | grep 2797 / ps -aux | grep

Display the stack trace for the `vsftpd` process.

-ps aux | grep 2797

Determine what files are currently opened by the `vsftpd` process.

socket:[10981]

Determine what system call `vsftpd` is currently executing.

Accept

Display the kernel-space stack trace for the `vsftpd` process.

top mview | less
for
vsftpd

& use psid f

Lab 7.3: Controlling Processes with the Magic SysRq Key

Scenario: When a process running on your system exhausts all CPU resources and blocks access to the console, your task is to force this process to stop.

Instructions:

1. Access the text mode console (*Ctrl-Alt-F1*) and log in as root. Switch your system to runlevel 3. ✓
2. Enable the Magic SysRq key so it is enabled persistently after a reboot. Enable it now and confirm it is working. ✓
3. Download the **multilock** program from the instructor machine. It is located in the `/var/ftp/pub/rhd361-materials` directory on the instructor's server. ✓
4. Execute **multilock**. Notice the console is now unresponsive. Switching to other consoles will not work either. The only way to get the system back to work is to reboot or force the **multilock** process to terminate.
5. Use the Magic SysRq key to inspect the system and to terminate the **multilock** process. ✓

Lab 7.1 Solutions

1. Make sure the `kernel-headers`, `kernel-doc` and `kernel-devel` packages are installed on your system.

```
[root@host ~]# yum install kernel-headers ✓  
kernel-doc kernel-devel
```

2. Install the `kernel-debuginfo-common` and `kernel-debuginfo` packages:

```
[root@host ~]# yum install -y kernel-debuginfo
```

Determine where the `vmlinux` file got installed:

```
[root@host ~]# rpm -ql kernel-debuginfo | grep ✓  
vmlinux
```

Lab 7.2 Solutions

1. Switch to the text mode interface (*Ctrl-Alt-F1*) and log in as root.

Determine the architecture and the version of the kernel that is currently running.

```
[root@host ~]# uname -a
```

2. Install the **vsftpd** package and start the **vsftpd** daemon. Make sure it restarts automatically when the system reboots:

```
[root@host ~]# yum -y install vsftpd
```

```
[root@host ~]# service vsftpd start
```

```
[root@host ~]# chkconfig vsftpd on
```

Determine the process identification number (PID) of the **vsftpd** daemon you just started:

```
[root@host ~]# pidof vsftpd
```

3. Display the memory map of the **vsftpd** daemon.

```
[root@host ~]# pmap $(pidof vsftpd)
```

Display the stack trace for the **vsftpd** process.

```
[root@host ~]# pstack $(pidof vsftpd)
```

Determine what files are currently opened by the **vsftpd** process.

```
[root@host ~]# ls -l /proc/$(pidof vsftpd)/fd/
```

Determine what system call **vsftpd** is currently executing.

```
[root@host ~]# strace -p $(pidof vsftpd)
```

Display the kernel-space stack trace for the **vsftpd** process.

```
[root@host ~]# echo t > /proc/sysrq-trigger
```

```
[root@host ~]# dmesg
```

You can also use **crash** to find the answers to some of these questions.

Lab 7.3 Solutions

1. Access the text mode console (*Ctrl-Alt-F1*) and log in as root. Switch your system to runlevel 3.

```
[root@host ~]# init 3
```

2. Enable the Magic SysRq key so it is enabled persistently after a reboot by editing `/etc/sysctl.conf`. Change the `kernel.sysrq` line to read as follows:

```
kernel.sysrq = 1
```

Execute the following command to enable the Magic SysRq key immediately:

```
[root@host ~]# sysctl -p
```

Type *Alt-SysRq-h* to confirm the Magic SysRq key is active.

3. Download the **multilock** program from the instructor machine. It is located in the `/var/ftp/pub/rhd361-materials` directory on the instructor's server.

```
[root@host ~]# cp -p \
/net/instructor/var/ftp/pub/rhd361-materials/mult
ilock .
```

4. Execute **multilock**. Notice the console is now unresponsive. Switching to other consoles will not work either. The only way to get the system back to work is to reboot or force the **multilock** process to terminate.
5. Press the *Alt-SysRq-t* key combination to see what tasks are currently running on the system. **multilock** should appear in the output displayed by the kernel.

Press the *Alt-SysRq-e* key combination to terminate all currently running processes (including **multilock**).





Lecture 8

Interrupts

Upon completion of this unit, you should be able to:

- Understand hard and soft interrupts
- Understand the role of tasklets
- Understand work queues



- Alter the sequence of instructions currently being executed
- Two types of interrupts:
 - Synchronous
 - Also called “exceptions”
 - Occurs as the result of a programmed instruction
 - Asynchronous
 - Usually called “interrupts”
 - Caused by I/O devices or timers

Synchronous interrupts, or exceptions, are generated within the CPU. They occur as the result of a programming error or when an exception occurs that requires the attention of the kernel, a page fault for example.

Asynchronous interrupts are generated by external hardware devices such as timers or I/O devices. Interrupt controllers allow many external devices to be connected to a limited number of pins on the CPU. We will discuss different types of hardware used to handle interrupts later in this unit.

```
[user@host ~]$ head -n 15 /proc/interrupts
CPU0
0: 93646257 IO-APIC-edge timer
1: 2511 IO-APIC-edge i8042
6: 5 IO-APIC-edge floppy
7: 0 IO-APIC-edge parport0
8: 1 IO-APIC-edge rtc
9: 1 IO-APIC-level acpi
12: 11403 IO-APIC-edge i8042
14: 30767 IO-APIC-edge ide0
15: 840478 IO-APIC-edge ide1
169: 85632 IO-APIC-level uhci_hcd:usb4, eth0
177: 0 IO-APIC-level ehci_hcd:usb1
185: 26936 IO-APIC-level uhci_hcd:usb2, eth1
193: 0 IO-APIC-level uhci_hcd:usb3
NMI: 0
```

Handwritten note: 2 uhci



- Interrupts can occur at any time
- One interrupt can be interrupted by an interrupt of another type
 - Objective is to keep all I/O devices busy (not serialize)
 - Some critical sections must disable interrupts
 - Minimize this as much as possible
- Programming goals for dealing with interrupts
 - Acknowledge the interrupt as soon as possible
 - Defer as much processing as possible

The functions used to manage interrupt requests (IRQs) include:

Function	Purpose
<code>local_irq_disable()</code>	Turn off local IRQs
<code>local_irq_enable()</code>	Turn on local IRQs
<code>local_save_flags(flags)</code>	Save current IRQ state flags
<code>local_irq_save(flags)</code>	Save current IRQ state and disable IRQs
<code>local_irq_restore(flags)</code>	Restore IRQ state

Older code may call the following functions to manage interrupts: `cli()`, `sti()`, `save_flags()`, `save_flags_cli()`, and `restore_flags()`. These functions are deprecated and should not be used in new code.



- Processor detected exceptions
 - Faults - correctable, instruction is re-executed
 - Traps - instruction not re-executed
 - Aborts - not correctable, current process terminates
- Programmed exceptions
 - System calls and debugger breakpoints
- Maskable interrupts
 - Normal timer and I/O events
- Non-maskable interrupts (NMI)
 - Usually a critical hardware failure

Faults are exceptions that are usually correctable. For example if a program tries to access a page of memory that has not been allocated, a page fault occurs and the program will not resume until the needed page is available. Once the kernel allocates the page and updates the processes memory space so the virtual address that caused the fault is valid, the instruction that caused the fault is re-executed.

Traps are exceptions that resume execution of the instruction immediately following the trap. Traps are usually used for debugging purposes, for example when a breakpoint is reached. Aborts are non-correctable, fatal exceptions that cause termination of the currently executing process.

Programmed exceptions occur when a program intentionally executes an instruction that raises a processor exception, such as `int`, `int3`, `into`, or `bound`. System calls are an example of programmed exceptions.

Maskable interrupts are associated with normal I/O device interrupt requests (IRQs). Unmasked IRQs are immediately recognized by the CPU when they arrive. Masked IRQs are ignored by the CPU until they are unmasked. Non-maskable interrupts (NMIs) usually occur when critical events, such as hardware failures, arise. NMIs are always recognized immediately by the CPU.



- Programmable Interrupt Controller (PIC)
 - Usually discrete devices separate from the CPU
 - Often found on uniprocessor systems
 - Two cascaded 8259A-type chips monitor up to 15 IRQs
- Advanced Programmable Interrupt Controller (APIC)
 - Can be an external device
 - Internal device on newer processors (local APIC)
 - Used in an SMP environment
 - Monitor up to 24 IRQs
- Interprocessor Interrupts (IPI)

When hardware wants to get the attention of the CPU, it raises a signal on one of its hardware lines which issues a interrupt request (or IRQ). Programmable interrupt controllers (PICs) on uniprocessor systems and advanced programmable interrupt controllers (APICs) on multiprocessor systems provide greater flexibility on how interrupts are reported to the CPU. First they consolidate many hardware lines into much fewer lines that go into the CPU. Interrupt controllers also provide more control over how interrupts are handled. They can defer, or mask, the handling of an IRQ, they can prioritize the order IRQs are reported to the CPU, and they can determine which processor is interrupted in an SMP environment.

Interprocessor interrupts (IPIs) allow CPUs in an SMP environment to exchange messages with each other. The most common messages sent between CPUs cause them to invoke the scheduler (`RESCHEDULE_VECTOR`) or rebuild their translation look-aside buffers (`INVALIDATE_TLB_VECTOR`). IPIs can be sent to all CPUs, the current CPU (*self*), all CPUs but *self*, or a specific set of CPUs specified in a mask.



- Maps each interrupt vector to its corresponding handler
- Three types of 8-byte descriptors (Intel hardware)
 - Task gate descriptor
 - Interrupt gate descriptor
 - Interrupts disabled
 - Linux application: asynchronous interrupts
 - Trap gate descriptor
 - Interrupts enabled
 - Linux application: exceptions

The interrupt descriptor table (IDT) is essentially a jump table for all kinds of interrupts: exceptions, faults, and IRQs. It contains 256 8-byte entries with the following fields:

- P - present bit, set when table entry is initialized
- Type - task gate, interrupt gate, or trap gate
- DPL - descriptor privilege level
- Remaining bits specify the address of the gate handler
 - Task gate - TSS segment selector of process to replace current one
 - Interrupt and trap gates - segment selector plus 32-bit offset to handler



- DPL field - descriptor privilege level
 - 0 - interrupt vector only accessible when in kernel mode
 - 3 - vector accessible from both kernel and user modes
- Linux interrupt descriptors
 - Task gate - Intel task gate with DPL = 0
 - Interrupt gate - Intel interrupt gate with DPL = 0
 - System interrupt gate - Intel interrupt gate with DPL = 3
 - Trap gate - Intel trap gate with DPL = 0
 - System gate - Intel trap gate with DPL = 3

The descriptor privilege level (DPL) bits of an IDT entry define the privilege level the handler will execute in. They are used by the processor handing an interrupt to perform security checks. If the current privilege of a process is greater than the DPL of the segment which contains the code of the exception handler, a general protection exception occurs because a handler cannot successfully execute with less privileges than the calling process. On the other hand, if the DPL bits in the IDT are less than the current privilege level of the process, this also causes a general protection fault. This prevents a lower privilege process from escalating priority because of an intentional exception.



- `setup_idt()` initializes default IDT entries at boot time
 - `Unused/uninitialized vectors point to ignore_int()`
 - Prints “Unknown interrupt” to console when called
- `trap_init()` initializes the IDT with standard exceptions
 - Interrupt vectors defined by the following functions:
 - `set_task_gate(n, addr)`
 - `set_intr_gate(n, addr)`
 - `set_system_intr_gate(n, addr)`
 - `set_trap_gate(n, addr)`
 - `set_system_gate(n, addr)`

The following code snippet from `trap.c` shows the IDT exceptions being initialized:

```
void __init trap_init(void)
{
... output omitted ...
    set_trap_gate(0, &divide_error);
    set_intr_gate(1, &debug);
    set_intr_gate(2, &nmi);
    set_system_intr_gate(3, &int3); /* int3/4 can be called from all */
*/
    set_system_gate(4, &overflow);
    set_trap_gate(5, &bounds);
    set_trap_gate(6, &invalid_op);
    set_trap_gate(7, &device_not_available);
    set_trap_gate(8, GDT_ENTRY_DOUBLEFAULT_TSS);
    set_task_gate(9, &tss);
... output omitted ...
    set_system_gate(SYSCALL_VECTOR, &system_call);
```

The `set_system_gate()` function is used to register the handler for interrupt 0x80 (`SYSCALL_VECTOR`). It has a DPL of 3 so all privilege levels can invoke it.



- Assembly language wrapper: `handler_name()`
 - For example: `divide_error()`
- High-level C exception handler: `do_handler_name()`
 - For example: `do_divide_error()`
- `do_trap()` sends a signal to the current process which generated the exception
 - Result:
 - User mode process executes a signal handler (if registered), or
 - Kernel takes default action (usually terminating the process)

The assembly language wrapper is architecture-specific code that is needed to properly interact with the IDT interface. It handles some basic error checking, and returns from the interrupt properly.

The high-level handler in C is architecture-independent code that raises the appropriate software signal for the process which generated the exception. If the user program registered a signal handler, then the signal handler will execute. Otherwise, the kernel will take the default action—in most cases, this terminates the process, which may then dump a core file.



- Three possible sources for asynchronous interrupts
 - I/O devices
 - Timers
 - Another CPU (IPIs)

Since IDT table entries 0 through 31 are reserved for use as processor exceptions, the remaining entries (32 through 255) are used for external interrupts. As we saw previously, entry 0x80 (interrupt 128) is the gate used for system calls. The following table lists the IDT assignments from 32 through 255:

IDT Entry	Function
32-127	External IRQs
128	System call vector (0x80)
129-238	External IRQs
239-240	Local APIC interrupts
241-250	Reserved by Linux
251-253	Inter processor interrupts (IPIs)
254-255	Local APIC interrupts

Note IRQ 0 maps to interrupt 32 in the IDT, not to interrupt 0 (which is mapped to an exception).



- Many devices can share the same vector (IRQ)
- A given handler can be activated by multiple devices of the same type
 - Code must be re-entrant
- Should be installed only when needed
 - For example when a device is opened, not when the module is loaded



- Interrupt descriptor structure fields:
 - `handle_irq`: high-level irq-events handler (if `NULL`, `__do_IRQ()`)
 - `handler_data`: per-IRQ data for the `irq_chip` methods
 - `chip`: low level interrupt hardware access
 - `chip_data`: platform-specific per-chip private data for the `chip` methods, to allow shared `chip` implementations
 - `action`: the irq action chain
 - `status`: flags describing IRQ line status
 - `depth`: disable-depth, for nested `irq_disable()` calls
 - `lock`: spin lock used for SMP
 - `irq_count`, `irqs_unhandled`: for diagnosing spurious IRQs



- The action field (`irq_desc` structure) points to a linked list of `irqaction`
- IRQ action structure fields:
 - `handler`: address of the interrupt service routine
 - `flags`: interrupt type flags
 - `IRQF_SHARED` - interrupt can be shared
 - `IRQF_DISABLED` - disable local interrupts while processing
 - `IRQF_SAMPLE_RANDOM` - use the interrupt for the entropy pool
 - `name`: I/O device name displayed in `/proc/interrupts`
 - `dev_id`: private device data
 - `irq`: IRQ line
 - `next`: pointer to next `irqaction`



- Request IRQ - allocate an interrupt line
 - `request_irq(irq, handler, irqflags, devname, dev_id)`
 - `irq`: interrupt line to allocate
 - `handler`: function to be called when the IRQ occurs
 - `irqflags`: interrupt type flags
 - `devname`: an ASCII name for the claiming device
 - `dev_id`: a cookie passed to the handler function
 - `request_irq()` calls `setup_irq()` to do the dirty work
- Free IRQ - remove a handler from an IRQ line
 - `free_irq(irq, dev_id)`
 - `irq`: Interrupt line to free
 - `dev_id`: device identity to free (originally passed to `request_irq()`)



- Softirqs
- Tasklets
- Work queues
- Four types of operations performed on deferred functions:
 - Initialization
 - Activation
 - Masking
 - Execution



- **Types of softirqs used in Linux 2.6:**
 - `HI_SOFTIRQ` - high priority tasklet handlers
 - `TIMER_SOFTIRQ` - timer event handlers
 - `NET_TX_SOFTIRQ` - network packet transmission handlers
 - `NET_RX_SOFTIRQ` - network packet reception handlers
 - `BLOCK_SOFTIRQ` - block device handlers
 - `TASKLET_SOFTIRQ` - regular priority tasklet handlers
 - `SCHED_SOFTIRQ` - scheduler handler for load balancing
 - `HRTIMER_SOFTIRQ` - high resolution timer (optional)
 - `RCU_SOFTIRQ` - read-copy-update
- **`softirq_action` Structure**
 - `action`: pointer to the softirq handler



- Initialized with `open_softirq()` function
- Activated by calling `raise_softirq()`
- Pending softirqs are performed (max 10 iterations)
 - at the end of asynchronous interrupts, by `irq_exit()`
 - after local APIC timer is handled
 - after local IPI is handled (SMP)
 - `ksoftirqd` kernel thread wakes up



- Tasklets are the preferred method for scheduling deferred work for I/O drivers
- Tasklets are actually implemented as softirqs
- `tasklet_vec` and `tasklet_hi_vec` arrays define low and high priority
- `tasklet_struct` Structure
 - `state`: current status of the tasklet
 - `TASKLET_STATE_SCHED` - tasklet is scheduled for execution
 - `TASKLET_STATE_RUN` - tasklet is running (SMP only)
 - `count`: lock counter
 - 0 = enabled, greater than 0 = disabled
 - `func`: address of the tasklet function
 - `data`: address passed as an argument to the function
 - `next`: pointer to the next tasklet in the linked list



- Structure initialized with `tasklet_init()` function
- Installed into the tasklet queue (by priority)
 - `tasklet_schedule()`
 - `tasklet_hi_schedule()`
- Enabled and disabled by the following functions
 - `tasklet_enable()`
 - `tasklet_disable()`
 - `tasklet_disable_nosync()`
- Executed when softirqs are performed



- Work queues are a mechanism for defining kernel helper threads for running arbitrary tasks in process context
 - Cannot make assumptions about which process is active
 - Therefore, cannot access user space addresses
- A predefined work queue called "events" available for all to use



- **cpu_workqueue_struct Structure**
 - **lock**: spinlock used to protect the structure
 - **worklist**: pointer to linked list of pending functions
 - **more_work**: wait queue used by worker threads when they sleep because they have nothing to do
 - **current_work**: pointer to the work_struct being executed
 - **wq**: pointer to workqueue_struct containing this entry
 - **thread**: process descriptor of the worker thread
 - **run_depth**: run_workqueue() recursion depth
- **work_struct Structure**
 - **data**: pointer passed to the pending function (and flags)
 - **entry**: pointers to next/previous entries in the queue
 - **func**: address of the pending function



- Create a work queue
 - `create_workqueue(name)`
- Insert a function into a work queue
 - `queue_work(wq, w)`
 - `queue_delayed_work(wq, w, d)`
 - `cancel_delayed_work_sync(w)`
- Wait until all work queue work has been completed
 - `flush_workqueue(wq)`

The following functions use the predefined `events` work queue:

- `schedule_work(w)`
- `schedule_delayed_work(w, d)`
- `schedule_delayed_work_on(cpu, w, d)`
- `flush_scheduled_work()`



End of Lecture 8

- Questions and Answers
- Summary
 - Difference in handlers for hard and soft interrupts



Lab 8.1: Introduction to Interrupt Handlers

Deliverable: A kernel module which installs an interrupt handler which counts interrupts raised on an IRQ

Instructions:

1. Write a kernel module called `rh361-irq` that installs an interrupt handler on an IRQ line that is specified as a parameter to the module when it is loaded. Make sure it can share the IRQ with other modules.

Your interrupt handler should count how many times it is called. Write it in such a way that it doesn't interfere with normal system operation. Do not call `printk()` from within your handler.

Have your kernel module print the number of times the handler was called when the module is unloaded.

2. Once you have a working module, test it by installing it on an IRQ that you know will generate some interrupts.

Lab 8.2: Bottom-Half Handling with Tasklets

Deliverable: Updated kernel module that installs an interrupt handler which creates a tasklet which counts interrupts raised on an IRQ

Instructions:

1. Write a kernel module called `rhd361-task` that installs an interrupt handler on an IRQ line that is specified as a parameter to the module when it is loaded. Make sure it can share the IRQ with other modules.

Your interrupt handler should count how many times it is called. Have your module do the counting within a tasklet instead of having the interrupt handler routine modify the counter. Write your code in such a way that it doesn't interfere with normal system operation.

Have your kernel module print the number of times the handler was called when the module is unloaded. Also print every 100th count from the tasklet.

2. Once you have a working module, test it by installing it on an IRQ that you know will generate some interrupts.

Lab 8.1 Solutions

1. Solutions for the programming exercises in this lab can be found on the instructor machine at `/var/ftp/pub/rhd361-solutions/08-interrupts`.
2. Once you have a working module, test it by installing it on an IRQ that you know will generate some interrupts. The best way to identify a likely IRQ candidate is to display `/proc/interrupts` and look for an IRQ with a high interrupt count.

Remember `request_irq()` will fail to register an interrupt handler on an IRQ that is already used by a handler that does not permit sharing of an IRQ.

Before you load your driver, display `/proc/interrupts` and note the number of IRQs for the IRQ you intend to have your driver count. Record the count when you unload your driver and compare the difference between the count after and before with the count your driver indicates.







Lecture 9

Device Driver Overview

Upon completion of this unit, you should be able to:

- Understand the inner workings of device drivers and how they interact with the kernel



- The software interface between hardware and the kernel
- Low-level, hardware-specific components enabling devices to interact with more generic, higher-level APIs
- Kernel integration options:
 - Compile directly in kernel
 - Always available for instant access
 - Increases the size of the kernel, possibly unnecessarily
 - Compile as a loadable kernel module
 - Access only when specific device is required
 - Incurs very small overhead to search for and load driver before the device can be accessed
 - Dynamic loading nature provides significant advantage during development and deployment

For more detailed information, see the follow-on to this course: *RHD362 - Kernel Internals 2: Device Driver Programming*.



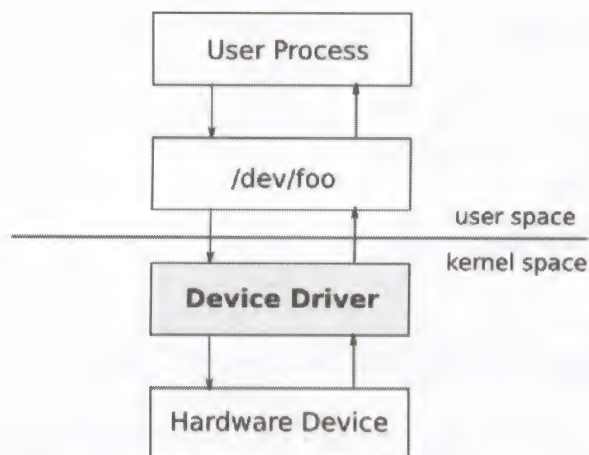
- Character
 - Unbuffered I/O
 - Sequential and random access
- Block
 - Accessed through a system cache
 - Sends and receives block-sized requests
 - Random access only
 - Filesystems required to use
- Network
 - No `/dev` directory entry point
 - Transmits and receives packets sent to it from the outside to the kernel

Character, block, and network device drivers are the main types of device drivers, though we'll limit our discussion to that of a character device driver. A computer's mouse, keyboard, and console are examples of character devices, whereas a disk drive or one of its partitions is an example of a block device.

A device driver is responsible for transferring data to and from the device. Character device drivers communicate directly with the calling user space application and no buffering of the data is performed. Block devices are instead accessed by user space applications through a system buffer that acts as a data cache. Random access to the data is permitted and since all I/O is buffered in a memory cache, allocation and memory management routines are not necessary during transfer of data to/from the device.



- User space applications communicate with device drivers through device nodes (files)



- "Everything is a file" model
- Located in `/dev` by convention
- Examples:

```
$ ls -l /dev/null /dev/zero
crw-rw-rw- 1 root root 1, 3 Jul  1 09:34 /dev/null
crw-rw-rw- 1 root root 1, 5 Jul  1 09:34 /dev/zero
```

A device driver is the interface between the user space application and the hardware. It is part of the kernel (whether compiled as a dynamically loadable module or statically compiled into it) and so does its work in kernel space.

While there exist user space device drivers, we won't discuss them here.

When a user space application wants to communicate with a hardware device, it communicates via a device file, which acts like a communications conduit, directly with the device driver. This follows the UNIX/Linux model of "everything is a file" (network devices are the exception). The operations that can be performed on the device are dictated by the device drivers file operations table.



- Device nodes are configured with a:
 - *name* - e.g. `/dev/foo`
 - *type* - either `b` (block device) or `c` (character device)
 - *major number* - identifies the device driver associated with the device
 - *minor number* - identifies specific device of possibly several using the same driver
- Creating a device node:
 - `mknod name type major minor`
 - Example: `mknod /dev/foo c 250 0`
- `Documentation/devices.txt`
- The name and file attributes can be affected by `udev`
 - `/etc/udev/rules.d/*.rules`

The type of device that the device node is a connector for is identified by the character in the first column in the output of the `ls -l` command. If it is a `c`, it is associated with a character device. If it is a `b`, it is associated with a block device.

Where the size of the file is usually displayed in the output of the `ls -l` command, a comma-delimited pair of numbers is displayed instead if it is a device node. The first number is the major number and the second number is the minor number. A major number is programmed into the device driver by its author and then registered with the kernel when loaded. The major number on the device node then identifies which device driver in the kernel this file is associated with. Because several device nodes may use the same device driver (for example, one each representing the different partitions on a SATA drive), the major number must be used in conjunction with a minor number to uniquely identify the device. No other device node may share the same combination of major and minor number.

An example of creating a device node:

```
# mknod /dev/foo c 250 0
# ls -l /dev/foo
crw-r--r-- 1 root root 250, 0 Jul  6 19:01 /dev/foo
```

For more information about allowable major numbers, see the file `Documentation/devices.txt`.



- Goal: when a driver's device node is accessed, the driver module is dynamically loaded
- For the following example:
 - `crw-r--r-- 1 root root 250, 0 Jul 6 19:01 /dev/foo`
- Use an entry in `/etc/modprobe.conf` similar to the following to load a character driver module named `foo_cdrv.ko`:
 - `alias char-major-250-0 foo_cdrv`
- Device driver module must be in the path searched by **depmod**

In the above example, we want character device driver module `foo_cdrv.ko` to be dynamically loaded whenever the `/dev/foo` device node is accessed. The corresponding entry in `/etc/modprobe.conf` specifies that any access of a device node with major number 250 and minor number 0 should load the character driver `foo_cdrv.ko` (notice the `.ko` extension is dropped in the entry).

In order to be found, the device driver must exist somewhere in the path searched by **depmod**.

The minor number (and preceding dash) can be left off the entry in `modprobe.conf` or an asterisk ("`*`") can be used in place of the minor number to indicate that only the major number is relevant in deciding to load the module.

For block devices a similar entry is used, replacing the "char" with "block" (e.g. `alias block-major-250-*`).



- Stored together in 32-bit *device number* of type `dev_t`
- The device number holds the major and minor parts:
 - 12 bits for major number
 - 20 bits for minor number
- Device drivers can share major numbers so long as the major/minor combination is unique
- From `<linux/kdev_t.h>` (where `kdev` is the device number):
 - `unsigned major = MAJOR(dev_t kdev);`
 - `unsigned minor = MINOR(dev_t kdev);`
 - `kdev = MKDEV(int major, int minor);`
- Major/Minor number assignation:
 - `int register_chrdev_region(dev_t kdev, unsigned int count, char *name);`
 - `void unregister_chrdev_region(dev_t kdev, unsigned int count);`

While it would seem that the maximum number of devices that could be handled by the operating system is represented by the number of bits in the `dev_t` variable ($2^{12} = 4096$), the number is actually much larger because device drivers can share major numbers, so long as the minor number or range of minor numbers used in combination with the major number are independent.

This was not true for the older 2.4 kernel, where a 16-bit `dev_t` variable was used (8 bits for the major number and 8 bits for the minor number), and there was a strict separation of major numbers between device drivers.

Use the macros above for the extraction of the major/minor number from the `dev_t` variable and also for creating it from a statically assigned major/minor combination so that future changes will encode/decode the values properly should the bit assignments change again.

In the `register_chrdev_region` function (`fs/char_dev.c`), the device number `kdev` is pre-created with `MKDEV` and pre-determined major and minor numbers. The minor number used for `kdev` is usually 0, and starts the range of minor numbers used for this device. The `count` parameter is a request for `count` contiguous minor numbers (too large and it will spill over to the next major number, if available). The `name` parameter is the name associated with this/these devices and appears in `/proc/devices` and `sysfs (/sys/devices/...)`.

The return value of `register_chrdev_region` and `unregister_chrdev_region` is 0 for success and a negative value for an error.



- The kernel can also create major and minor numbers for your device on the fly
- Assigning a dynamic device number range:
 - `alloc_chrdev_region(dev_t *kdev, unsigned int baseminor, unsigned int count, const char *name);`
- `kdev` is populated with the first device number if successful
- Still use `unregister_chrdev_region` to free range when no longer in use

Often times you don't know what major number to use, or even care to know. In these cases, a major number can be dynamically assigned by the kernel using the `alloc_chrdev_region` function.

If successful (return value of 0, like other kernel functions), `alloc_chrdev_region` populates `kdev` with the first device number in the requested range.

The next question then becomes "How do I ensure my device file's major number matches that of my device driver?" We answer that in the next slide.



- Utilizes a struct class device pointer
- In the initialization fxn, create the struct class pointer...
 - `static struct class *new_class;`
 - `new_class = class_create(THIS_MODULE, "example_class");`
- ...then create the device and register it with sysfs:
 - `device_create(x_class, NULL, kdev, "%s%d", "example", 1);`
 - The first device would be named `/dev/example1`
- When the device is no longer in use, clean up in the exit fxn:
 - `device_destroy(x_class, kdev);`
 - `class_destroy(x_class);`

Functions used:

```
struct class *class_create(struct module *owner, char *name)
struct device *device_create(struct class *myclass, struct device *parent,
    dev_t devt, char *fmt, ...)
void device_destroy(struct class *myclass, dev_t devt)
```

The `owner` refers to which module "owns" this struct class, and is usually set to `THIS_MODULE`. The `name` is a pointer to a string used to refer to the class. `myclass` is the pointer to the struct class that this device should be registered to. `parent` can be a pointer to a parent struct device of this new device, and is set to `NULL` if there is none. `kdev` is the `dev_t` for this character device, and `fmt` is used for the name of the initial device and succeeding devices (`%d`, initially set to 1, in this case).



- The `miscdevice` interface is a simple way to manage device nodes
- Define a `miscdevice` structure
 - `name` field is the name of the device node
 - `fops` field points to file operations structure
 - `minor` field is requested minor device number (or `MISC_DYNAMIC_MINOR`)
- Initialization function calls `misc_register()`
 - Which creates the character special device
- Exit function calls `misc_deregister()`
 - Which removes the character device

Functions used:

```
int misc_register(struct miscdevice *misc);
int misc_deregister(struct miscdevice *misc);
```

The `miscdevice` structure has the following definition:

```
struct miscdevice {
    int minor;
    const char *name;
    const struct file_operations *fops;
    struct list_head list;
    struct device *dev;
    struct class_device *class;
};
```

The only fields which have to be defined before calling `misc_register()` are:

- `name` - this string determines the name of the device node created in `/dev`
- `fops` - this points to the file operations structure which handles IO to the device
- `minor` - the minor number to be used for the IO device file

The `minor` field can have a positive integer value assigned to it or the value `MISC_DYNAMIC_MINOR` which allows the `miscdevice` interface to dynamically assign its own minor value. When the minor number is dynamically assigned, the actual value used is returned in the `minor` field when `misc_register()` returns. The `misc_register()` function returns 0 on success, negative `errno` on failure. Note the structure whose address is passed to `misc_register()` should be persistent until `misc_deregister()` is called.



- All things true about *Kernel Module Essentials* still apply here
- Additional typical `#include`'s:
 - `linux/fs.h`: File table structures and operations
 - `linux/uaccess.h`: The `copy_(to|from)_user` functions
 - `linux/slab.h`: Various memory-related utilities (`kmalloc`)
 - `linux/cdev.h`: Various `cdev` utilities

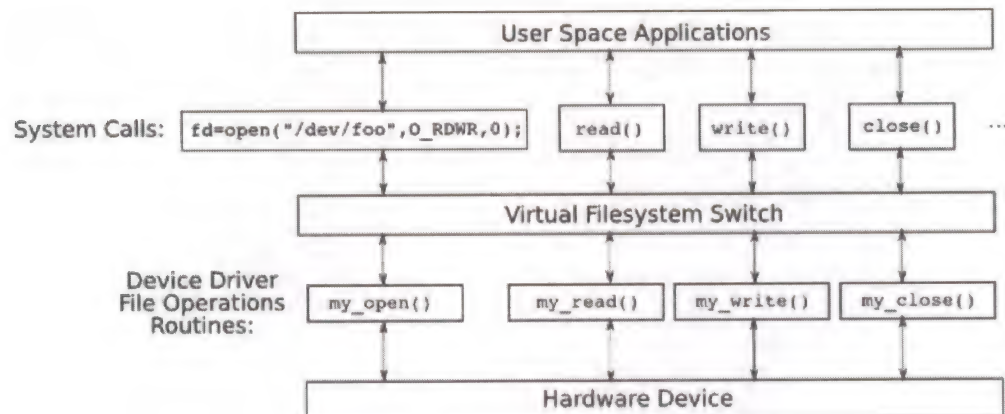


- Before the kernel invokes a device operation, a structure must be created to represent it
- `cdev` structure from `<linux/cdev.h>`

```
struct cdev {  
    struct kobject kobj;  
    struct module *owner;  
    const struct file_operations *ops;  
    struct list_head list;  
    dev_t dev;  
    unsigned int count;  
};
```

- Include the proper header file:
 - `<linux/cdev.h>`
- Allocate memory for and initialize the structure, then set the file operations structure for it:
 - `struct cdev *mychrdrv = cdev_alloc();`
 - `cdev_init(mychrdrv, &fops);`
- The driver is enabled and disabled with:
 - `cdev_add(mychrdrv, kdev, count);`
 - `cdev_del(mychrdrv);`

See `fs/char_dev.c` and `<linux/cdev.h>` for more information.



In order to talk to the kernel, the driver registers with subsystems to respond to events. Such an event might be the opening of a file, a read/write, a page fault, the plugging in of a new USB device, etc.

When registering to handle an event, a file operations structure is initialized at load time within the device driver to specify how each file operation will be handled. The kernel uses this file operations structure (fops) to access the driver's functions. For example, a `read()` operation on the file `/dev/foo` in the example above would invoke the `my_read()` function within the device driver identified by the major number of `/dev/foo`. The `my_read()` function within the device driver might in turn read data from the hardware device it was written to manage.



- Driver methods (*entry points*) are defined in the `file_operations` structure
 - The list of operations an application can invoke on a device: `<linux/fs.h>`
 - `__user` annotation
 - System calls to device file are passed to corresponding method
 - Associated with device via `cdev_init()` function
 - `owner` is the only one that is not a method
- Example `file_operations` structure:

```
struct file_operations fops = {  
    .owner      = THIS_MODULE,  
    .open       = my_open,  
    .read       = my_read,  
    .write      = my_write,  
    .release    = my_close,  
};
```

- If no entry point is supplied for a system call, either:
 - It will fail (e.g. `mmap()`)
 - A default method will be invoked (e.g. `open()` and `release()`)

Driver methods (also called *entry points*) are called by the device driver whenever the corresponding system call is made on its associated device file. We use object-oriented terminology to represent the *object* (the device file) and the *method* (the function operating on it).

The full list of operations an application can invoke on a device are listed in `<linux/fs.h>`. Looking through the list, you will notice several prototype parameters that are described with `__user`. The `__user` annotation indicates (as a form of documentation) that a pointer is a user-space address that can't be directly dereferenced from kernel space. `__user` has no effect on compilation.

For example, a device driver associated with `/dev/example` and with a `fops` structure of (C99 syntax used here):

```
struct file_operations fops = {  
    .owner      = THIS_MODULE,  
    .open       = my_open,  
    .read       = my_read,  
    .write      = my_write,  
    .release    = my_close,  
};
```

the `my_open` function is called whenever `/dev/example` is opened by an application in user space. The `my_read` function is called whenever an application tries to read from `/dev/example` (e.g. **cat /dev/example**), etc. A `fops` field is left `NULL` for unsupported operations. How the kernel behaves when a `NULL` pointer is specified is dependent upon the function being invoked.

The full list of driver entry points (`kernel-2.6.18-92.el5`):

```
struct file_operations {
```



```

struct module *owner;
loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
int (*readdir) (struct file *, void *, filldir_t);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *, fl_owner_t id);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, struct dentry *, int datasync);
int (*aio_fsync) (struct kiocb *, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area) (struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
    int (*check_flags) (int);
    int (*dir_notify) (struct file *filp, unsigned long arg);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write) (struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
    ssize_t (*splice_read) (struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
};

```



- Kernel space programs, only (not the one in libc)
- A new structure is created upon `open()` of a device
 - Memory and structure are released upon `release()`
- Passed to all functions that use the device node
- There can be multiple `file` structures per device
- Kernel space programs
 - Not for use in user space applications
- `include/linux/fs.h`

The `file` data structure is unrelated to the libc `FILE` data structure, and is not used in user space programs. If a device supports multiple `open()`s, there can be multiple `file` structures for it at the same time.

```
struct file {
    /*
     * fu_list becomes invalid after file_free is called and queued via
     * fu_rcuhead for RCU freeing
     */
    union {
        struct list_head    fu_list;
        struct rcu_head     fu_rcuhead;
    } f_u;
    struct dentry            *f_dentry;
    struct vfsmount          *f_vfsmnt;
    const struct file_operations *f_op;
    atomic_t                 f_count;
    unsigned int             f_flags;
    mode_t                   f_mode;
    loff_t                   f_pos;
    struct fown_struct        f_owner;
    unsigned int             f_uid, f_gid;
    struct file_ra_state     f_ra;

    unsigned long            f_version;
    void                     *f_security;

    /* needed for tty driver, and maybe others */
    void                     *private_data;

#ifdef CONFIG_EPOLL
    /* Used by fs/eventpoll.c to link all the hooks to this file */
    struct list_head         f_ep_links;
    spinlock_t               f_ep_lock;
#endif /* #ifdef CONFIG_EPOLL */
    struct address_space     *f_mapping;
};
```



- include/linux/fs.h
- Only one inode structure per device node
- Each open descriptor on the device node points to the inode structure

```
struct inode {
    struct hlist_node    i_hash;
    struct list_head     i_list;
    struct list_head     i_sb_list;
    struct list_head     i_dentry;
    unsigned long        i_ino;
    atomic_t             i_count;
    umode_t              i_mode;
    unsigned int         i_nlink;
    uid_t                i_uid;
    gid_t                i_gid;
    dev_t                i_rdev;
    loff_t               i_size;
    struct timespec      i_atime;
    struct timespec      i_mtime;
    struct timespec      i_ctime;
    unsigned int         i_blkbits;
    unsigned long        i_version;
    blkcnt_t             i_blocks;
    unsigned short       i_bytes;
    spinlock_t           i_lock; /* i_blocks, i_bytes, maybe i_size */
    struct mutex          i_mutex;
    struct rw_semaphore   i_alloc_sem;
    struct inode_operations *i_op;
    const struct file_operations *i_fop; /* former ✓
->i_op->default_file_ops */
    struct super_block    *i_sb;
    struct file_lock      *i_flock;
    struct address_space  *i_mapping;
    struct address_space  i_data;
#ifdef CONFIG_QUOTA
    struct dquot          *i_dquot [MAXQUOTAS];
#endif
    struct list_head     i_devices;
    ...fields omitted...
    atomic_t             i_writecount;
    void                 *i_security;
    void                 *i_private; /* fs or device private pointer ✓
*/
#ifdef __NEED_I_SIZE_ORDERED
    seqcount_t           i_size_seqcount;
#endif
};
```




- Typical open method tasks:
 - Initialize the device (if first time opened)
 - Check for and handle device-specific errors (e.g. device not ready)
 - Allocate and populate private data structures
 - Identify the device being opened (the minor number)
- Typical release method tasks:
 - Undo anything open did (e.g. deallocate resources created)
 - Suspend/Shutdown device if no longer being used



- read copies data from device to application memory
 - `ssize_t read(struct file *filp, char __user *userbuf, size_t count, loff_t *offp);`
- write copies data to device from application memory
 - `ssize_t write(struct file *filp, const char __user *userbuf, size_t count, loff_t *offp);`
- The return values are the number of bytes successfully transferred
 - Negative return values indicate an error as defined in `<linux/errno.h>`
- Helper functions:
 - `unsigned long copy_to_user(void __user *to, const void *from, unsigned long n)`
 - `unsigned long copy_from_user(void *to, const void __user *from, unsigned long n)`
- `offp` should be updated after a successful read/write to represent the current file position

Keep in mind there are several methods available for transferring data between user and kernel space. Here we discuss the simplest methods.

In the above prototypes:

- `filep`: a pointer to the file
- `userbuf`: a user-space pointer to a buffer holding data to be written or the (empty) buffer a read will populate with data
- `count`: number of requested bytes to be transferred
- `offp`: a pointer indicating the file's position of access
- `to`: a pointer to the destination address, in kernel (`copy_from_user`) or user (`copy_to_user`) space
- `from`: a pointer to the source address, in kernel (`copy_to_user`) or user (`copy_from_user`) space
- `n`: number of bytes to copy

The file position offset pointer (`offp`) should be updated after a successful read or write to represent the current file position. The kernel will propagate the position information back into the file structure as needed.



- The kernel must know how many user-space processes are referencing a module
- Only when the usage count is zero can the module be unloaded
- The kernel tracks this count automatically if the module's `owner` field is set in the appropriate data structure for the module
- The `owner` entry point:
 - Not an operation, but a pointer to the module that *owns* the structure
 - Almost always initialized to `THIS_MODULE`
 - Refers to, literally, *this* module
 - Macro defined in `include/linux/module.h`
- Manually changing a module's usage count
 - `int try_module_get (struct module *module);`
 - `void module_put (struct module *module);`
 - No effect if `CONFIG_MODULE_UNLOAD` is not set *usr/src/linux/.config*
- The count is maintained in the module data structure
 - `unsigned int module_refcount (struct module *mod);`

The module usage count has nothing to do with how many times it is being used by another module, as found in the output of `lsmod`.

There exist other structures with similar entry point (callback function) `owner` fields, such as `file_operations`, `block_device_operations`, and `fb_ops`. For example, the following might be used within a block device module:

```
static struct block_device_operations myblk_drv_fops = {
    .owner = THIS_MODULE,
    ...
};
```

The `owner` field of the structure is not an operation field, but rather is a pointer to the module that owns the structure, and almost always is set to `THIS_MODULE`.

For example, the current usage count can be printed with the following line:

```
printk("User-Space Process Reference Count = %d\n",
    module_refcount(THIS_MODULE));
```




```
MODULE_DESCRIPTION("RHD361 Sample Driver");
MODULE_LICENSE("GPL");

static ssize_t rhd361_read(struct file * file,
                           char __user * buf, size_t len, loff_t *ppos)
{
    size_t todo = len;

    while (todo) {
        size_t chunk = (todo > 4096) ? 4096 : todo;

        if (clear_user(buf, chunk))
            return -EFAULT;
        buf += chunk;
        todo -= chunk;
        cond_resched();
    }
    return len;
}
```

The first part of the driver we look at is the code that implements the `read()` functionality. The `clear_user()` function clears `chunk` number of bytes of user-space memory pointed to by the address stored in `buf`. The call to `cond_resched()` allows this driver to voluntarily relinquish the CPU if there are other processes that are runnable. This is done since our device is a virtual device without hardware-based latencies.

The following are the header files that were omitted from the above slide for brevity:

```
#include <linux/module.h>
#include <linux/miscdevice.h>
#include <linux/kernel.h>
#include <linux/uaccess.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <linux/errno.h>
```



```
static ssize_t rhd361_write (struct file *file,
                             const char __user *buf, size_t len, loff_t *ppos)
{
    return len;
}

static loff_t rhd361_lseek(struct file * file,
                           loff_t offset, int orig)
{
    return file->f_pos = 0;
}

static struct file_operations rhd361_fops = {
    .owner      = THIS_MODULE,
    .read       = rhd361_read,
    .write      = rhd361_write,
    .llseek     = rhd361_lseek,
};

static struct miscdevice rhd361_misc_dev = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = "rhd361",
    .fops       = &rhd361_fops,
};
```

This slide contains the functions that implement the `write()` and `lseek()` methods for this device. Normally the write operation would copy data from the user-space buffer provided and perform the IO necessary to move the data to the device. In this case the data simply gets ignored which means this device acts as a bit bucket when written to. The `lseek()` function always resets the pointer to an offset of zero since no actual data is stored.

The code on this page also shows the population of the `file_operations` and `miscdevice` structures. The `name` field in the `miscdevice` structure will be used to create the character special device file when this module is loaded, in this case the file will be called `/dev/rhd361` and the minor number will be dynamically allocated since the `minor` field is `MISC_DYNAMIC_MINOR`.



```
static int __init rhd361_init (void)
{
    int error;

    error = misc_register(&rhd361_misc_dev);
    if (error) {
        printk(KERN_ERR "RHD361: can't misc_register "
                "on minor=%d\n", RHD361_MINOR);
        return error;
    }
    printk(KERN_INFO "RHD361 Example Driver loaded\n");
    return 0;
}

static void __exit rhd361_exit (void)
{
    misc_deregister(&rhd361_misc_dev);
}

module_init(rhd361_init)
module_exit(rhd361_exit)
```

Finally the module initialization and exit functions are presented. It is here the driver is registered and unregistered as a miscellaneous character driver by the `misc_register()` and `misc_deregister()` functions respectively. If `misc_register()` returns with a non-zero return value indicating an error, the driver will fail to load.



End of Lecture 9

- Questions and Answers
- Summary

- Different device types

Device nodes

Dynamically loading devices

Major and Minor numbers

Important device driver structures and methods





Lab 9.1: Writing a Read-Only Character Device Driver

Instructions:

1. Write a character device driver which does not accept writes and returns a fixed string when `read()`. The fixed string can be specified as a module parameter when your driver is loaded, but you should hard code a default string in case one isn't specified.

Lab 9.2: Writing a Read/Write Character Device Driver

Instructions:

1. Write a character device driver which internally uses a static 4KiB buffer as storage media which is initially zeroed. Writes to the device should be stored in the device and reads should return the data that was originally written. Reads should not consume the data, so another reader should retrieve the same data as a previous reader until another writer writes new data to the device.

Only one process should have exclusive read or write access to your device at a time. Also make sure your driver doesn't allow the user to write past the end of the device.

Lab 9.1 Solutions

1. Solutions for this lab are located on the instructor machine at `/var/ftp/pub/rhd361-solutions/09-devdriver`. The source code for these exercises are located in the above directory with a `Makefile` which will compile them appropriately.





Memory Management

Upon completion of this unit, you should be able to:

- Understand how virtual addresses are converted to physical addresses
- Understand how Linux organizes the virtual address space
- Distinguish between various memory allocators
- Know how to request memory from the appropriate allocator



- *Paging* provides isolation and protection of address space
 - Used to map physical memory to a virtual address space
 - Each process has a private section of physical memory for code and data
 - Processes can not see physical memory which is not mapped into their own virtual address space
 - *Shared memory* is physical memory mapped into the virtual address space of multiple processes at the same time
- Managed by hardware Memory Management Unit (MMU)
- Address space is limited by processor architecture

Modern processor architecture implement a virtual memory mechanism called *page translation* or *paging* to allow operating systems to more securely and flexibly manage system memory.

With paging, each process has its own private virtual memory address space. Physical memory is mapped into this virtual address space in fixed-size units called *pages*. A process can not see a region of physical memory unless it has been mapped into its virtual address space. This protects the memory space of one process from another.

Two or more processes may choose to share memory by mapping the same physical memory addresses to their own virtual address spaces. This *shared memory* may be writable by both processes, in which case it can be used as a communication channel. Alternatively, it can be provided read-only; for example, seven processes that need to use the same library functions could access the library code through shared memory. Instead of needing seven copies of the library stored in physical memory, there needs to be only one copy in physical memory which is then mapped by all seven processes into their respective virtual address spaces, read-only.

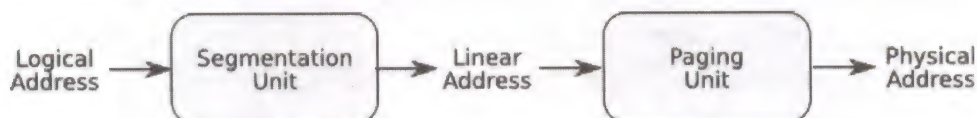
Likewise, the contents of a physical memory mapping can be transferred ("paged out") to disk, and when a process tries to access that virtual address, the contents can be transferred back to physical memory and remapped on the fly ("paged in").

Paging is managed in hardware by the processor's Memory Management Unit (MMU). The size of a page, the virtual address space which is used and the physical address space which is allowed depends on the processor architecture.

A 32-bit system allows a 2^{32} byte (4 GiB) virtual address space, and a 64-bit system allows a 2^{64} byte (16 EiB) virtual address space, although for various reasons the entire virtual address space may not be usable/mappable to physical memory. (For example, the x86-64 architecture as currently implemented in hardware only allows the use of 256 TiB of the 64-bit virtual address space for CPU design reasons.)



- Logical Addresses
 - Offsets into a memory *segment*
- Linear Addresses
 - Logical address adjusted by segment base address
- Physical Addresses
 - Bus address after page table translation



Memory management on the 32-bit x86 processor architecture and related 64-bit x86-64 architecture is complicated by the long and convoluted history of the design. Early x86 processors supported only segmentation, which divided physical memory into multiple regions, or segments. The *logical addresses* of these segments were originally mapped directly to physical memory addresses. Paging, based around 4 kiB pages and a flat memory model, was introduced with the Intel 80386 processor. As we've mentioned, paging allows each process to have its own virtual address space and enables pages to be "paged out" to swap space. Paging has always been the preferred mechanism which Linux uses to manage memory on all architectures. (It is not a coincidence that Linux was first developed on the Intel 80386 processor.) However, for legacy reasons on x86 and x86-64 segmentation still exists; now segmentation maps logical addresses to *linear (virtual) addresses* which must be mapped to physical addresses in hardware.

On x86/x86-64, a logical address is an offset into the program's segment. These logical addresses are mapped by hardware into a virtual linear address. Each process has its own virtual address space of linear addresses; 4 GiB on x86, 16 EiB (not all of which is usable on current processors) on x86-64. The paging unit then maps each logical address in use to a physical address in the actual memory hardware.



- *Segmentation* is an old memory management technique which breaks memory up into regions
 - Memory is addressed by specifying a segment, then the *segment offset* within that segment to access
 - We need to know the base address of the segment to convert this into an actual memory address
 - Obsoleted by paging, many newer architectures do not support segmentation (but 32-bit x86 *requires* it)
- Linux and other modern systems avoid using segmentation
 - Sets up one segment that covers all of virtual memory
 - *This makes the segment offset the same as the virtual address*

Segmentation is an old memory management technique which divides physical or linear memory into regions called *segments*. Memory is addressed by specifying the base address of the desired segment, followed by a *segment offset* within that segment. This allows code and data to be relocated to any part of memory, and protection of one segment from another.

Segmentation as a technique has been replaced in most modern processor architectures by paging and a flat memory model. For the 32-bit x86 architecture, segmentation is a legacy feature carried over from the Intel 80286 and earlier processors.

Linux generally avoids using segmentation, at least in part because many of the non-x86 architectures it runs on do not support segmentation at all. On 32-bit x86, segmentation can not be turned off; Linux sets segmentation up such that segments almost always map all of linear memory, and paging is used instead for protection and memory management purposes.



- A logical address has two parts: *segment base address* and *segment offset*
 - Information about segments is stored in a *Global Descriptor Table* (GDT)
 - The *segment register* in use points to a *segment descriptor* in the GDT
 - The segment descriptor contains the starting linear address of the segment, the ending linear address of the segment, and the minimum privilege ring allowed access to the segment
 - $\text{segment base address} + \text{segment offset} = \text{linear address}$
- Linux *always* sets segment base address to zero
 - Essentially does not use segmentation
 - Required by x86-64 in any case

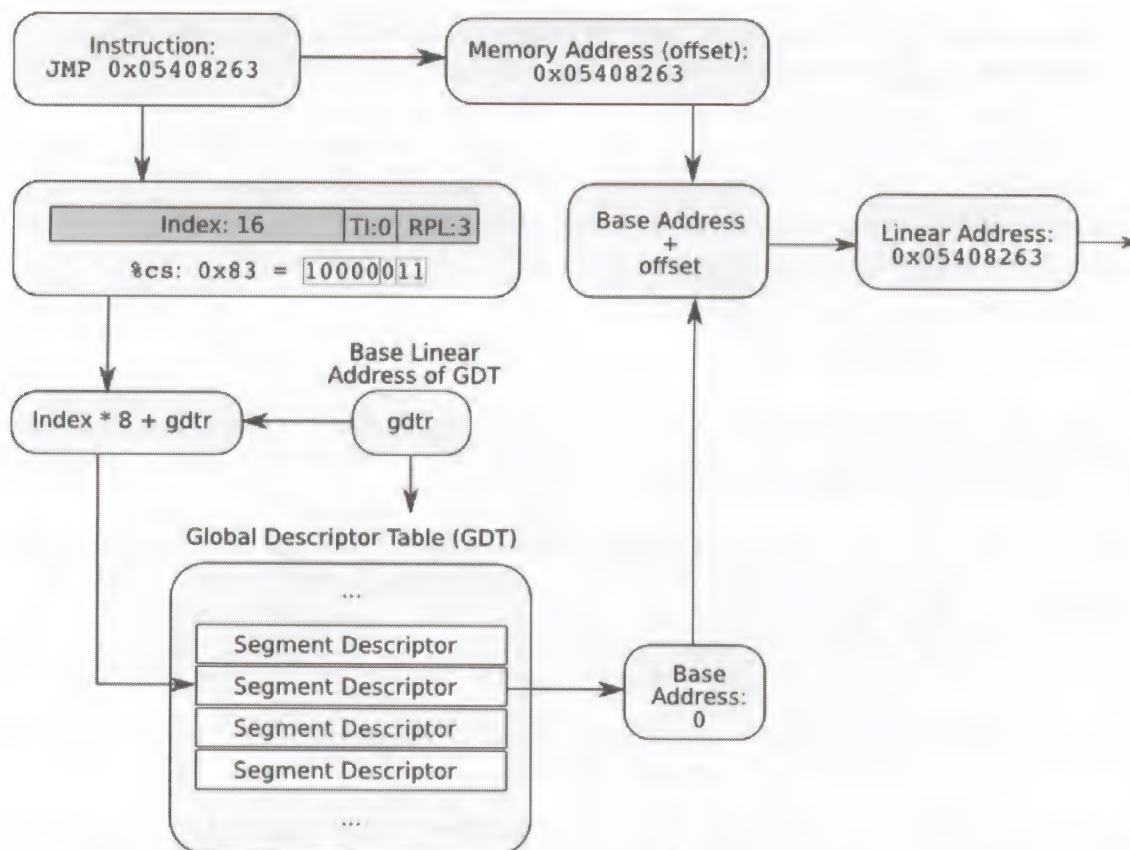
On the x86 architecture, a logical address is made up of two parts; a *segment base address* and a *segment offset*. The segment base address is the linear address of the start of the segment; the segment offset is the offset from the start of the segment that the desired address is at.

However, while a machine instruction starts with the segment offset, it must look up the segment base address. Information about the each segment, including its starting and ending address, is stored in a *segment descriptor* in a *Global Descriptor Table* (GDT). To find our current segment descriptor, the processor finds the location of the GDT from a register (`%gdt_r`), and looks in the current *segment register* (generally `%cs` for code or `%ds` for data) to find the index of the segment descriptor we need from the table.

Then, determining the linear address we need is simply a matter of adding the 32-bit segment base address from the segment descriptor to the 32-bit segment offset.

Linux avoids using segmentation on 32-bit x86, but this mechanism can not be turned off. Instead, it sets up two pairs of segments: code and data for kernel space (ring 0) and user space (ring 3), and *always* sets the base address to `0x00000000` and the end of the segment (the *segment limit*) to `0xFFFFFFFF`, thus creating segments that cover all of memory, and for which the segment offset is identical to the linear (virtual) address.

This is a common approach in modern operating systems which prefer paging and a flat virtual memory model in place of segmentation. The 64-bit x86-64 architecture makes this approach mandatory by forcing the segmentation base address to *always* be treated as `0x0000000000000000`, by ignoring segment limits, and by making the segment offset a 64-bit address rather than a 32-bit one.



The diagram above demonstrates how a 32-bit x86 processor converts a logical address to a linear (virtual) address.

In the example, the processor wants to execute the instruction `JMP 0x05408263`. The value `0x05408263` is the 32-bit *segment offset* from the start of the segment. The `%cs` register indicates that we want to look for information about the segment (the *segment descriptor*) at index 16 (`0x10000`) in the GDT, and we are currently running in privilege ring 3 (userspace).

The `%gdtr` register stores the base linear address of the GDT in bytes; we add the the index (16) multiplied by 8 bytes (the size of each segment descriptor) to the base linear address of the GDT to find the linear address of the segment descriptor that `%cs` is pointing to.

The segment descriptor is an 8 byte structure that stores the 32-bit base address of the start of the segment. This is *always* set to zero (`0x00000000`) by the Linux kernel! Therefore *the segment offset is actually the same as the linear address* the processor will use, simplifying programming immensely. The segment descriptor also indicates the ring protection level we need to access this segment (presumably ring 3 in this example).

The 20-bit segment limit in the segment descriptor indicates the end of the segment in pages; for Linux this is almost always page `0xFFFFF` (allowing access through linear address `0xFFFFFFFF`), meaning the segment is the same size as the entire linear address space. The only situation where this is not true is on 32-bit non-PAE systems with ExecShield enabled, which may set the segment limit for `%cs` lower to restrict processes from executing data above the segment limit as code. (Systems with PAE or running the

x86-64 architecture have NX support which can restrict access on a per-page basis, and will always set the segment limit to the end of virtual memory.)

Thankfully, segmentation is essentially deprecated for the x86-64 architecture in 64-bit long mode, and a simple flat linear addressing model is always used instead. In the 64-bit AMD64 architecture, the segmentation offset now is a 64-bit address. The size of the `%cs` register is unchanged (but also has a flag that indicates we are running in 64-bit long mode). The `%gdt_r` register still stores the base linear address of the GDT in bytes, but it is now a 64-bit register and address. Each segment descriptor is still 8 bytes in size. In normal 64-bit long mode, the processor *ignores* the base address indicating the start of the segment which stored in the descriptor, and *always* assumes it is actually `0x0000000000000000`; it also ignores the segment limit, effectively treating it as `0xFFFFFFFFFFFFFFFF`. Therefore the segment is always the size of the entire (64-bit) linear address space, no matter what. This means that, *for 64-bit x86-64, again, the segment offset is the same as the linear address* the processor will use.

The "bottom line": in Linux, on 32-bit x86 and 64-bit x86-64, the segment offset is always the same as the linear address. Segments cover the entire linear (virtual) address space on the system, unless the system is 32-bit non-PAE and ExecShield is in use—in which case the `%cs` segment limit will be set for a process such that all executable code is located below that address in memory.



- Virtual memory (linear addresses) are grouped into *pages*
- Physical memory is divided into *page frames*
- The contents of linear address pages can be mapped into page frames or stored in swap space
- Basic page size is determined by a processor's architecture
 - x86/x86-64 is 4 kiB
 - ppc64 is 64 kiB in RHEL 5, 4 kiB in RHEL 4 and earlier
 - `getpagesize()` system call returns the page size
- Some architectures support *hugepages* which require special handling

Modern systems support paging, a mechanism which allows the system to map a large virtual address space for each process to a (potentially) smaller physical address space. Linear addresses are grouped into contiguous fixed-size blocks called *pages*, which the operating system can move from physical memory (divided into page-sized *page frames*) to block device storage (swap space).

The size of a page depends on the design of the paging unit in the processor architecture. For x86/x86-64, the basic page size is 4 kiB. For 64-bit POWER (ppc64), RHEL 5 takes advantage of the 64 kiB page size available on recent POWER5+ and POWER6 processors. (Earlier versions of Red Hat Enterprise Linux used a 4 kiB page size for ppc64.) Larger page sizes generally provide better performance (through fewer TLB misses and less page fault overhead) at the cost of worse internal fragmentation in caches containing small objects.

Often, a special large page or *hugepage* size is also supported, which requires special handling by the operating system and has various limitations (must be preallocated by a system administrator through `hugetlbfs`, hugepages can not be paged out or demand-paged from a file on disk, etc.). For example, 32-bit x86 in PAE mode and x86-64 support a 2 MiB hugepage size, while ppc64 supports a 16 MiB hugepage size.



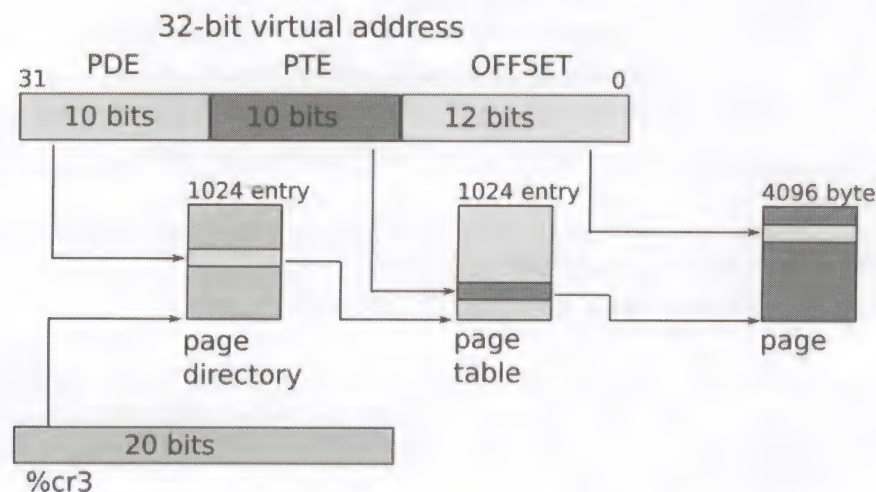
- Each process has its own set of page tables
 - Page table entry translates a page of virtual memory to a physical page frame (or swap index)
 - A hierarchy of page tables and directories is used to reduce memory usage
 - Only need to allocate memory for page directories or page tables containing mapped pages
- Virtual to physical page mappings cached in hardware by a *Translation Look-aside Buffer (TLB)*
 - Speeds performance; if we look up a second virtual address on the same page, can get mapping from a TLB instead of through lookups in page tables
 - On x86/x86-64, **x86info -c** shows size of instruction and data TLBs

Since each process maintains its own private virtual address space, each process needs its own mappings of virtual addresses to physical addresses. As we have seen, these mappings are made in blocks of contiguous 4 kiB pages. Each process therefore needs a hierarchy of *page tables* to store these translations (as *page table entries*).

For systems with 32-bit addresses and a 4 kiB page size like 32-bit x86, you can think of the memory address as having two parts. The high 20 bits (31-12) can be thought of as the *page number*, a number which uniquely identifies the page in memory. The low 12 bits (11-0) represent the offset of the byte within the page; for example, an offset of zero would be the first byte in the page. This works because a page is 4 kiB, or 2^{12} bytes, in size.

A single flat page table could be used per process, but even on a 32-bit x86 system this table would need 2^{20} 32-bit page table entries, which would require allocating 4 MiB of memory *per process*, even if a given process did not need to use all 4 GiB of its virtual memory pages. Instead, Linux divides the page table into a hierarchy of page directories, that point to smaller page tables, that contain page table entries. By doing this, a process only needs to allocate memory for its page tables that contain pages in use, and for its page directories that point to page tables that contain pages in use. Since pages tend to be allocated in contiguous regions of virtual memory, this reduces the amount of memory that needs to be allocated to the page table since the system does not need to allocate memory for all possible page directories and page tables. We will shortly discuss this mechanism in more detail.

When a translation from a virtual address to a physical address is looked up by the process, it is cached in hardware in a *Translation Look-aside Buffer*, or TLB. This special processor cache speeds up address translation by caching page mappings that we have recently used. Since memory tends to be accessed sequentially, that means that if we access two addresses on the same page, the second access can get the virtual to physical address mapping from the TLB rather than by performing a lengthy lookup process in the page table hierarchy, which improves performance. The size of the TLB can be determined on the x86 and x86-64 processor architectures with the **x86info -c** command. When a context switch happens and the current process is replaced by a new process, the CPU's TLBs are invalidated and the cache is considered empty again so that we use the virtual to physical mappings in the new process's page table hierarchy.



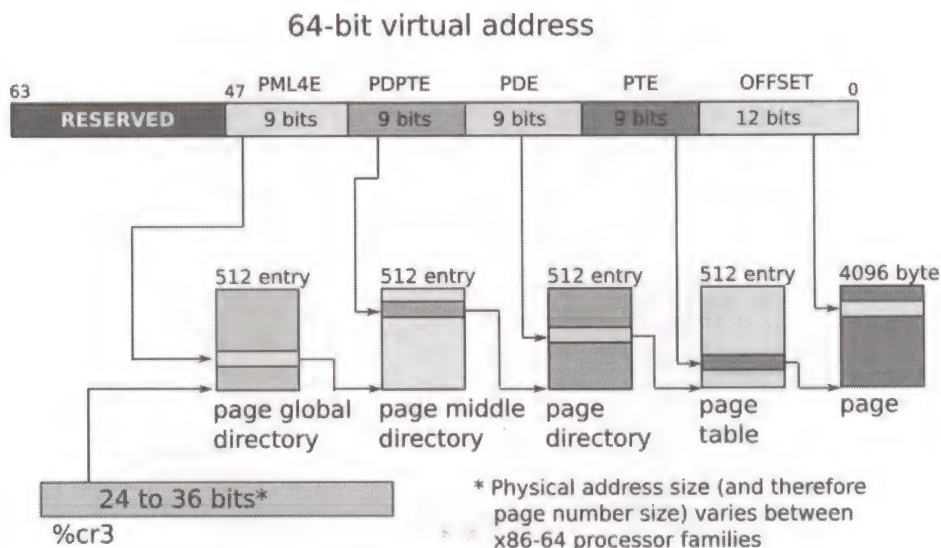
On 32-bit x86 without PAE, to translate a virtual address into a physical address, first we check the high 20 bits to see if the mapping is cached in the TLB. If it is, we can then access the page frame and use the low 12 bits of the virtual address to find the offset within the page frame that our data is at.

If our page mapping is *not* cached by the TLB, then we have to look it up in the page table hierarchy. The hardware register `%cr3` contains the 20 bit physical page number of the process's *page directory* which points to all the page tables which belong to this process. This directory contains 1024 entries (*page directory entries*, or PDEs) which point to page tables. The top 10 bits (31-22) of the virtual address point us to one of the 1024 PDEs within the page directory.

The PDE pointed to contains the 20 bit physical page number of the actual page table we need. Each page table maps 1024 pages; the next 10 bits (21-12) of the virtual address point us to the *page table entry* (PTE) that maps the virtual page to the physical page.

The PTE usually points to the 20 bit physical page number of the page we need. Then the low 12 bits of the virtual address (11-0) point to the offset within the physical page that our data is at. But the PTE may have the "Not Present" flag set for the page. If the rest of the PTE is blank, then the page is not allocated and we have a page fault. If the rest of the entry is not blank, Linux determines that the page has been swapped out. On 32-bit x86, the kernel uses five bits of the non-blank page table entry to indicate which *swap area* contains the page, and 24 are used to indicate the page slot in the swap area that contains the page. This allows there to be up to 32 swap areas of up to 64 GiB each.

For non-PAE x86, PDEs and PTEs are 32 bits wide; in addition to the physical page number stored in the entry and the flag which indicates if the page is in memory or swap, other flags exist in the entry indicating if the page is read-only or read-write, whether processes running in ring 3 can access the page, and so on. For x86 systems in PAE mode and x86-64, these entries are 64 bits wide—in addition to a larger physical page number, the wider entries also hold the NX/XD flag that indicates if the page is executable or not (which is used, if available, by ExecShield in place of segment limits to protect the system from buffer overflow attacks using code injection).

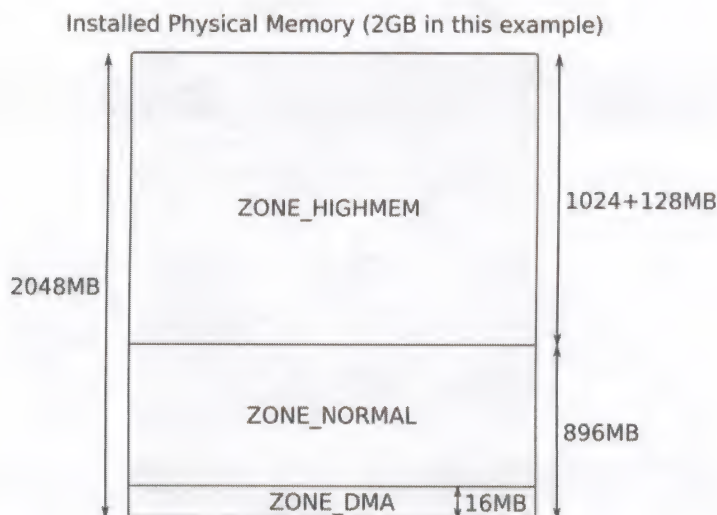


On 64-bit x86-64, we need a deeper page table hierarchy to control memory consumption by the larger virtual addresses. Also, in the current x86-64 processor design, only 48 bits of the 64 available in the linear address are used, giving us a 256 TiB virtual address space, which further simplifies the problem.

A four-level hierarchy is now used. The `%cr3` register now points to the base physical page number of the top-level *page global directory* for the process. Note that on x86-64, the size of the physical address supported by the CPU currently ranges from 36 bits to 48 bits, depending on the processor family, with 40 bits (1 TiB) being relatively common. (This physical address size results in a page number range which varies between 24 to 36 bits on different CPUs, with 12 bits of page offset.) This places an absolute limit on the amount of RAM that processor can support. The `address sizes` line in `/proc/cpuinfo` should indicate the true physical address size used on your particular machine.)

Bits 47-39 of the linear address point to the entry in the PGD which contains the base physical page number of the *page middle directory* (sometimes called the "page directory pointer table"). Bits 38-30 of the linear address point to the entry of the PMD that contains the base physical page number of the page directory. Bits 29-21 of the linear address point to the PDE that contains the base physical page number of the final page table, and bits 20-12 point to the entry on the page table that maps to the physical page. As before, bits 11-0 of the linear address point to the offset within the physical page of our data.

(32-bit x86 with PAE ("Physical Address Extensions") uses a similar mechanism to map 36-bit physical memory addresses into a 32-bit linear address space. It uses a 4-entry page global directory (using the first two bits of the virtual address, 31-30), pointing to 512-entry (9-bit offset) page directories, pointing to 512-entry (9-bit offset) page tables pointing to 4 kiB pages (12-bit offset). There is no page middle directory. Like x86-64, x86 PAE uses 64-bit entries rather than 32-bit entries in the page table hierarchy. The way this allows the 32-bit linear address to be mapped to 36-bit physical addresses is that `%cr3`, the PGD entries, the PDEs, and the PTEs all are modified to provide 24-bit physical page numbers instead of 20-bit physical page numbers. So while processes use 32-bit linear addresses, the system can theoretically have a 36-bit (64 GiB) physical address space. In practice, other memory limitations in kernel space tend to restrict usable physical memory on 32-bit x86 to 16 GiB.)

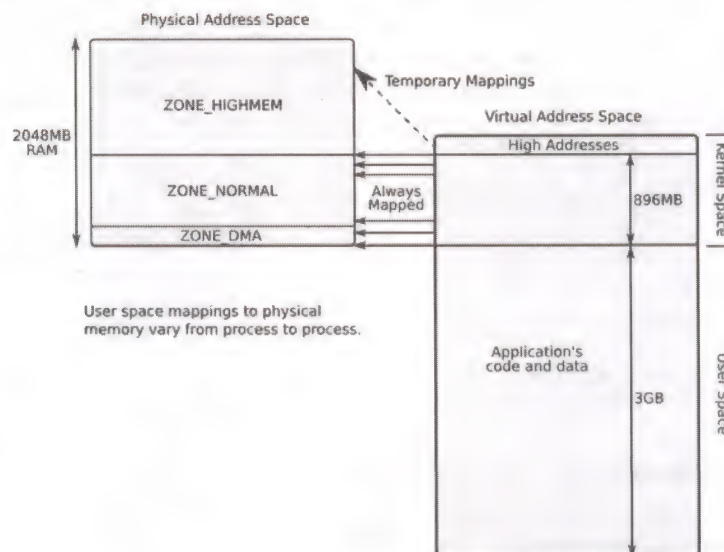


The kernel divides physical memory into *zones* based on various restrictions on how it may be used. On 32-bit x86, there are three zones: `ZONE_DMA`, `ZONE_NORMAL`, and `ZONE_HIGHMEM`.

The most important of these zones is `ZONE_NORMAL`. The kernel attempts to directly map all of physical memory into its linear (virtual) address space. On 64-bit x86-64 systems, or 32-bit x86 systems with less than 896 MiB of physical memory, this is possible.

In this simple case, all memory in use by a userspace process is mapped at least twice; once in the kernel's virtual address space as part of the permanent mapping of `ZONE_NORMAL`, and once from a virtual address in userspace for the use of the process.

On 32-bit x86, `ZONE_HIGHMEM` consists of physical memory in excess of 896 MiB that is harder for the kernel to access as it cannot be permanently directly mapped. Likewise, `ZONE_DMA` is code used by legacy devices for DMA operations that can only address the low 16 MiB of physical memory.



Each process has its own virtual address space. Process virtual address space is divided into two sections; *kernel space* and *userspace*. Linear addresses assigned to kernel space are used for kernel code, data structures, and memory mappings only, and those pages are protected so that they are only accessible when processes are executing kernel code (in ring 0). Linear addresses assigned to userspace are used for user code, data structures, and memory mappings (running in ring 3). All processes share the same kernel space virtual memory mappings, so that kernel code is available to service system calls and interrupts all the time. But their user space mappings differ, containing process specific data, and therefore user space mappings change on every context switch.

Kernel space starts in virtual memory at a linear address referred to as `PAGE_OFFSET` and ends at the top of the virtual memory space. `PAGE_OFFSET` is mapped to the start of physical memory.

For 32-bit x86, `PAGE_OFFSET` is `0xC0000000` (3 GiB), so kernel virtual memory is 1 GiB in size, running from `0xC0000000` to `0xFFFFFFFF`. Virtual address `0xC0000000` is mapped to physical address `0x00000000`, `0xC1000000` to `0x01000000`; this is a direct offset. Ideally, the kernel should be able to directly map any physical memory address into its virtual address space. Then the kernel can easily see and access all of physical memory. But remember that the 32-bit x86 kernel can only map up to 896 MiB of physical memory directly on 32-bit x86; it saves 128 MiB of its virtual address space for temporary dynamic mappings to addresses in high physical memory. This hack is required so that those physical addresses are reachable by the kernel.

Likewise, for 32-bit x86, userspace has virtual memory addresses `0x00000000` to `0xBFFFFFFF`, giving processes the ability to map (and therefore use) up to 3 GiB of memory. This allows processes to work with larger data sets in memory, even though many of the virtual addresses must map to physical addresses in highmem.

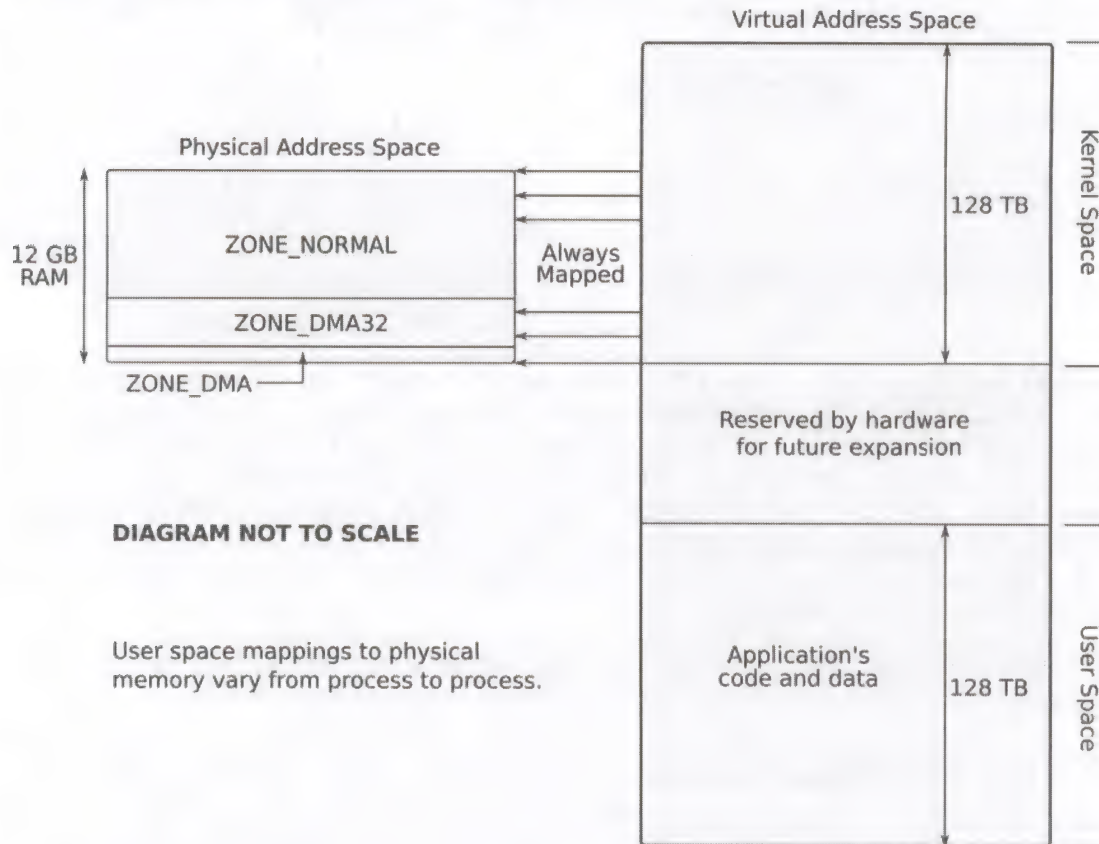
Note that, as already discussed, physical addresses will often have *multiple* linear addresses mapped to them from virtual memory of a process. For example, a physical address in `ZONE_NORMAL` will be permanently mapped by the kernel in its part of the virtual address space of the process, and may also be mapped by the process in the userspace part of its virtual address space for the use of the application.

The situation on 64-bit x86-64 is in many ways much more straightforward. `PAGE_OFFSET` for the RHEL 5 x86-64 kernel is `0xFFFFF81000000000`, which gives the kernel roughly 128 TiB of virtual address space.

Since the current RHEL 5 kernel supports at most 1 TiB of physical address space on x86-64, this is more than sufficient to directly map all of physical memory into the kernel region of virtual address space.

Userspace on 64-bit x86-64 is set up a bit differently on x86-64 as well. Linear addresses `0x0000800000000000` to `0xFFFF7FFFFFFFFFFFFF` are reserved by the hardware and may not be mapped to physical memory in currently produced x86-64 processors for hardware design reasons. Therefore, the virtual userspace mappings are confined to linear addresses from the start of memory to `0x00007FFFFFFFFFFFFF`, a 128 TiB region. (Again, this means that on x86-64 virtual userspace is much larger than the amount of physical memory supported by the hardware which could be mapped to it.) Note that this also means, unlike the 32-bit x86 architecture, on the 64-bit x86-64 architecture `PAGE_OFFSET` is *not* a sharp boundary between userspace and kernelspace.

Figure 10.1. x86-64 virtual address space





- Memory in `ZONE_NORMAL` can be directly mapped by the kernel in its part of the linear address space
- The `mem_map[]` structure at the bottom of `ZONE_NORMAL` tracks how physical memory is used
 - Rule of thumb for x86: about 11 MiB needed to manage each 1 GiB RAM
- The rest of `ZONE_NORMAL` is free for dynamic allocations by the kernel

Memory which the kernel has direct mappings for in its address space is in `ZONE_NORMAL`. This is also sometimes referred to as "lowmem" in the x86 architecture. This zone of memory is important because it is the most efficient to use and many kernel operations require memory allocated from this zone.

On a modern 64-bit system, or on a 32-bit system with modest amounts of physical memory (less than 896 MiB for the normal kernel), the kernel can map the entire physical address space directly. But on 32-bit x86, the 1 GiB size of the virtual address space for the kernel places a limit on how large `ZONE_NORMAL` can be; in this case, 896 MiB. Memory above that on 32-bit x86 is considered "highmem" and is more difficult to use.

The kernel keeps a `mem_map[]` structure which it places just above `ZONE_DMA` which tracks how physical memory is used. This structure itself consumes low memory, at about 11 MiB per GiB of RAM, and the memory pressure this places on `ZONE_NORMAL` is one reason why 32-bit x86 is limited to 16 GiB of physical memory even in PAE mode.



- On 32-bit x86 systems, the kernel is not able to directly map all of physical memory
 - Kernel reserves only 1 GiB of virtual address space
 - Only has room to directly map 896 MiB of memory
- Remaining 128 MiB of kernel virtual memory is used for temporary mappings to "highmem"
 - `kmap()` and `kunmap()` functions map/unmap from this region to highmem
- 64-bit x86-64 *can* directly map all of physical memory and has no `ZONE_HIGHMEM`

On 64-bit systems, it is easy to map all of physical memory inside the large linear (virtual) address space of the kernel. For example, on x86-64, where 48 bits of virtual address space is usable on current processors, kernel space is just under 128 TiB in size. This is far more than the 1 TiB of physical address space currently supported on that architecture in RHEL 5.

On 32-bit systems, the story is different. On x86, kernel space is only 1 GiB of virtual address space in size, much smaller than the 4 GiB of physical address space or 16 GiB of physical memory we support in RHEL 5 through PAE. This means that not all of physical memory can be directly mapped.

To deal with this, the x86 kernel only directly maps the low 896 MiB of physical memory. It then uses much of the remaining 128 MiB of its virtual address space (the *vmalloc area*) to set up temporary mappings to the "high memory" addresses above the 896 MiB line.

The function `void* kmap(struct page *pg)` is used to set up these transient mappings; to release the mappings the function `void* kunmap(struct page *pg)` is used. If you want to allocate a page and you do not mind that it is from `ZONE_HIGHMEM`, you can use the function `alloc_page(__GFP_HIGHMEM)` to request it.

`ZONE_HIGHMEM` is a hack which is necessary to get around the limitations of the 32-bit x86 architecture when dealing with modern amounts of physical memory. It makes working with code and data in the kernel harder and less efficient. One of the advantages of 64-bit systems is that they allow us to deal with large amounts of physical memory without resorting to the `ZONE_HIGHMEM` workaround.



- Used for older devices which cannot perform DMA to any address in the entire physical address space
 - For example, on the x86/x86-64 architectures ISA-based devices only supported the low 24 bits for addressing
 - To do 24-bit DMA, must use physical memory in the first 2^{24} bytes = 16 MiB
- On x86/x86-64, use `ZONE_DMA` to get memory in the low 16 MiB
 - x86-64 also has a `ZONE_DMA32` for devices supporting 32-bit DMA that must use physical memory from the first 4 GiB
- Linux tries to conserve its use of `ZONE_DMA` so devices will find free DMA memory if needed

Direct Memory Access (DMA) is a mechanism which allows devices to read and write from system memory without involving the CPU. This is an important feature because it can allow the system to transfer data rapidly to block devices and video cards without involving the CPU in the details of the I/O transfer.

However, certain devices and system buses have limits on the regions of memory they may address through DMA. For example, the obsolete ISA bus only could address the low 24 bits of physical memory. Therefore, to allow such devices to perform DMA, the kernel would have to ensure that memory allocated for such purposes came from the low 16 MiB of physical memory. Some legacy devices (serial ports, floppy disk controllers, parallel ports, PS/2 ports) may also fall into this category.

On both x86 and x86-64 architectures, `ZONE_DMA` is used to request low memory suitable for 24-bit DMA from the kernel. The newer 64-bit x86-64 architecture must also handle PCI devices that are 32-bit DMA capable but not 64-bit DMA capable. Therefore, x86-64 also has a `ZONE_DMA32` for these devices. The 32-bit x86 architecture does *not* have the `ZONE_DMA32` zone; `ZONE_NORMAL` can be used for this purpose on a 32-bit architecture.

Devices which can use the entire address space of the architecture for DMA may simply use memory allocated from `ZONE_NORMAL`.



- Kernel per-process stack size is too small to use big buffer
 - Kernel stack size is 4 kiB for x86/x86-64 in RHEL 4 and later kernels
- Kernel offers several different types of dynamic allocation functions to programmers
 - Physically contiguous or non-contiguous
 - Size-based
 - Objective
 - General vs. Specific



- Kernel manages dynamic memory using the data structures for page frame handling
 - Per page frames
 - Buddy System algorithm
 - Per small object
 - Slab Allocator
 - New algorithms: Slob and Slub
 - Non-contiguous memory area
 - `vmalloc()`



- Kernel manages all free page frames using buddy algorithm
- All free pages are grouped at boot time by kernel
- Allocate large block of contiguous page frames without using paging circuitry
- View available, contiguous segments of physical memory:

```
cat /proc/buddyinfo
```

```
echo m > /proc/sysrq-trigger
```



- General functions for requesting and releasing page frames
 - `alloc_pages()`
 - `__get_free_pages()`
 - `__free_pages()`
 - `free_pages()`
 - ...

When requesting page frames, the kernel attempts to avoid fragmentation by allocating pages in a contiguous block. When releasing frames, the kernel attempts to merge adjacent frames together into a contiguous block.



- Buddy System algorithm is not particularly efficient
 - If request size is less than a page in size, it wastes space by requesting whole pages
 - If create and destroy small same-size objects
- Slab allocator is an algorithm to manage 'objects' into caches
 - Cache objects
 - Avoid internal fragmentation



- The following functions are general functions for requesting and releasing slab objects
 - `kmalloc()`
 - `kfree()`
- Functions to directly use Slab Allocator
 - `kmem_cache_create()`, `kmem_cache_destroy()`
 - `kmem_cache_alloc()`, `kmem_cache_free()`
- Display slab memory
 - `/proc/slabinfo`
 - **slabtop**
 - **vmstat -m**



- Allocate memory based on non-contiguous page frames accessed through contiguous linear addresses
 - Linear addresses are contiguous
 - Physical memory does not guarantee contiguous memory
 - Bad performance
- Using high-memory mapping for x86



- Functions for requesting and releasing non-contiguous memory:
 - `vmalloc()`
 - `vfree()`
- Display `vmalloc`'s memory
 - `/proc/meminfo`

The anatomy of the Linux VM Allocator



- Mask of the following flags
 - `__GFP_DMA`, `__GFP_HIGHMEM`, `__GFP_WAIT`,
- Common MACROS (combination above flags)
 - `GFP_KERNEL`
 - `__GFP_WAIT | __GFP_IO | __GFP_FS`
 - `GFP_ATOMIC`
 - `__GFP_HIGH`
 - `GFP_DMA`
 - `__GFP_DMA`



- General purpose memory management function using Buddy System
- Physically contiguous
- To allocate entire pages of memory (2^{order})
- Function
 - `long __get_free_pages(gfp_mask, order)`
 - `void __free_pages(addr)`



- General Purpose memory management function using Slab Allocator
- Physically contiguous
- To allocate memory Object from slab cache
 - Less than a page in length
- Limitation
 - x86/x86_64 RHEL4/RHEL5
 - 128 kiB (32 pages)
 - x86/x86_64 upstream(>2.6.2x)
 - 4 MiB (1024 pages)
- Function
 - void *kmalloc(size_t size, gfp_t gfp_mask)
 - void kfree(const void *objp)



- The previous memory allocation functions are good for general purpose small allocations
- For larger memory allocations, the Linux Kernel offers another method: `vmalloc()`
- Theoretically there is no limitation of allocation size, with one restraint:
 - Needs free memory areas that are managed in Kernel
 - Mapping space in kernel space (too small on 32-bit architecture)
- Limitation
 - Not ensure physically contiguous
 - The penalty of access speed
 - Possible to sleep!
- Function
 - `void *vmalloc(unsigned long size)`
 - `vfree(addr)`



End of Lecture 10

- Questions and Answers
- Summary
 - Segmentation
 - Page Tables
 - DMA, Normal, and High Memory Zones
 - Buddy Allocator
 - Slab Allocator
 - `kmalloc()` and Friends





Lab 10.1: Allocating Kernel Dynamic Memory

Instructions:

1. Write a character device driver which maintains a dynamic linked list of all processes which have opened it. Within the `open()` function, allocate a dynamic structure:

```
struct pinfo {  
    struct list_head f_list,  
    pid_t    pid,      # current process pid  
    uid_t    uid,      # current process uid  
    char*    comm,     # current process   
    command  
};
```

where the `pid`, `uid`, and `comm` are copied from the current process' `task_struct`. Attach the structure to a linked list of all concurrent open files for this driver.

On reads, return a buffer which contains information about all of the processes using this driver from the linked list of `pinfo` structures. (Note: the string manipulation is the hard part. :()

Make sure the linked list of `pinfo` structures is appropriately protected. 

2. Write a user-space program to access the device file passed as an argument to the program. Open the device file, then repeatedly read from it and display what was read every two seconds. Keep the device file open so your process continues to be attached to the driver.
3. Bonus Exercise: In the returned information, have the driver identify which process is the process currently doing the reading. 

Lab 10.2: Using a Custom Slab Allocator

Instructions:

1. Modify the kernel module in the previous lab exercise to use the slab allocator to create a custom cache to manage the `pinfo` structures.
2. Use user-space tools to monitor the system's caches, paying particular attention to the behavior of your module's cache as the device driver is used.

Lab 10.1 Solutions

1. Solutions for the programming exercises in this lab can be found on the instructor machine at `/var/ftp/pub/rhd361-solutions/10-memory`.







Lecture 11

Processes

Upon completion of this unit, you should be able to:

- Describe how processes are created and destroyed
- Examine the memory layout of a process
- Describe what happens during a context switch
- Describe when context switches occur in a preemptive and non-preemptive kernel.



- New processes are created using one of the system calls:
 - `clone()`
 - `fork()`
 - `vfork()`
- The child process gets a logical copy of the parents resources (memory areas, file descriptors, signal handlers, environment)
- Parent and child processes read the same physical pages
- When either the parent or child write to the pages, the contents are copied to a new page (copy-on-write)
- Some resources can be shared between the child and parent process
- Changes made by either parent or child are visible to both
- If resources are shared, these are "lightweight" processes
- If not, these are "heavyweight" processes

There is actually a `clone()` C library call, as well:

```
#include <sched.h>
```

```
int clone(int (*fn)(void *), void *child_stack,  
int flags, void *arg, ...  
/* pid_t *pid, struct user_desc *tls, pid_t *ctid */ );
```

which actually is a wrapper around the `clone` system call, which does not have the `fn` and `arg` parameters.

```
_syscall2(int, clone, flags, void *child_stack)
```

The `clone()` call will begin executing the function `fn` with arguments `*arg`.



- Resources shared between parent and child processes are determined by the `flags` parameter
- Low byte of `flags` specifies the termination signal sent to the parent when the child dies
- `fork()` is implemented as `clone()` with `flags=SIGCHLD`
 - No resources are shared and the child signals the parent with `SIGCHLD` when it terminates
 - Initially the child's stack is the same as the parents stack
 - The child continues execution at the next instruction after the `fork`
 - The stack is copied when one process tries to change it (copy-on-write)
- `vfork()` is implemented as `clone()` with `flags=SIGCHLD` or `CLONE_VM` or `CLONE_VFORK`
 - Legacy call where parent's execution is blocked until the child calls `execv()` or exits
 - Not used under normal circumstances

The `flags` parameter can have the following values (see `clone(2)` for details):

- `CLONE_PARENT`: If set, then the parent of the new child will be the same as that of the calling process. When the process terminates, the parent of the calling process will be signaled.
- `CLONE_FS`: If set, the caller and the child processes share the same file system information.
- `CLONE_FILES`: If set, the calling process and the child processes share the same file descriptor table.
- `CLONE_NEWNS`: Start the child in a new namespace.
- `CLONE_SIGHAND`: If set, the calling process and the child processes share the same table of signal handlers.
- `CLONE_PTRACE`: If specified, and the calling process is being traced, then trace the child also.
- `CLONE_UNTRACED`: If specified, then a tracing process cannot force `CLONE_PTRACE` on this child process.
- `CLONE_STOPPED`: If set, then the child is initially stopped and must be resumed by sending it a `SIGCONT` signal.
- `CLONE_VFORK`: If set, the execution of the calling process is suspended until the child releases its virtual memory resources via a call to `execve()` or `_exit()`
- `CLONE_VM`: If set, the calling process and the child processes run in the same memory space.
- `CLONE_THREAD`: If set, the child is placed in the same thread group as the calling process.
- `CLONE_SYSVSEM`: If set, then the child and the calling process share a single list of System V semaphore undo values.



- Kernel space functions that call `do_fork()` (`kernel/fork.c`):
 - `sys_fork()`
 - `sys_clone()`
 - `sys_vfork()`
- `do_fork()` does the following:
 - Allocates a new PID for the child by looking in the `pidmap_array` bitmap
 - Invokes the `copy_process` function which returns the address of the new `task_struct`
 - Adjusts the scheduling parameters of the parent and child
 - Inserts the child in the parent's run queue
 - Returns the PID of the child



- The kernel represents a process's user space memory area with a `mm_struct` memory descriptor. For any process, this is defined as the `mm` field in the process's `task_struct`
- The `mm_struct` which is (mostly) detailed below actually points to a list of memory areas - which are intervals of memory addresses which the process has permission to access. The process can dynamically add or remove memory areas through the kernel. (We'll describe how later)
- The memory areas are represented in two ways in the `mm_struct`. First, by a linked list `*mmap`, which is used for operations that traverse the whole address space, and secondly, by a red-black tree - `mm_rb`, which is used for search, insert and delete operations. The semantics of a red-black tree ensure that these operations are $O(\log n)$

```
struct mm_struct {
    struct vm_area_struct * mmap;           /* list of VMAs */
    struct rb_root mm_rb;                   /* rb tree of VMAs */
    struct vm_area_struct * mmap_cache;     /* last find_vma result */
    unsigned long mmap_base;                /* base of mmap area */
    unsigned long task_size;                /* size of task vm space */
    unsigned long cached_hole_size;         /* the largest hole */
    unsigned long free_area_cache;          /* first hole */
    pgd_t * pgd;                            /* page global directory */
    atomic_t mm_users;                      /* How many users ? */
    atomic_t mm_count;                      /* How many references */
    int map_count;                          /* number of VMAs */
    struct rw_semaphore mmap_sem;
    spinlock_t page_table_lock;             /* Protects page tables */

    /* total number of pages for all, locked, shared, executable */
    unsigned long total_vm, locked_vm, shared_vm, exec_vm;

    /* total number of pages in stack, not swappable, default access flags,
       page table entries */
    unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;

    /* start and end addresses of code, data, heap and stack */
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;

    /* start and end arguments of arguments and environment */
    unsigned long arg_start, arg_end, env_start, env_end;
```



- Each memory area is represented by a `vm_area_struct`, which is shown below. The `vm_area_struct` points back to the associated `mm_struct` through `*vm_mm` and to the next memory area in the linked list through `vm_next`
- `vm_start` is the lowest address in the memory interval, and `vm_end` is the first byte after the interval.
- The `vm_ops` field points to a table of operations that can be executed on this memory area - specifically `open()`, `close()`, `page()`, `populate()`. These will be discussed later.
- `vm_flags` is a bit mask that specifies the behavior of the pages contained in the memory area. This includes read, write and execute permissions and whether the memory area is shared.

```
struct vm_area_struct {
    struct mm_struct * vm_mm;           /* The address space we belong to. */
    unsigned long vm_start;             /* Our start address within vm_mm. */
    unsigned long vm_end;               /* First byte after end address */

    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next;

    pgprot_t vm_page_prot;              /* Access permissions of this VMA. */
    unsigned long vm_flags;             /* Flags, listed below. */
    struct rb_node vm_rb;

    struct list_head anon_vma_node; /* Serialized by anon_vma->lock */
    struct anon_vma *anon_vma;        /* Serialized by page_table_lock */

    /* Function pointers to deal with this struct. */
    struct vm_operations_struct * vm_ops;

    /* Information about our backing store: */
    unsigned long vm_pgoff;             /* Offset in PAGE_SIZE units */
    struct file * vm_file;              /* File we map to (can be NULL). */
    void * vm_private_data;             /* was vm_pte (shared mem) */
    unsigned long vm_truncate_count; /* truncate_count or restart_addr */
};
```



Some of the most important `vm_flags` are listed below. They are all defined in `linux/mm.h`

- `VM_READ`, `VM_WRITE`, `VM_EXEC` define the access permissions of pages in this memory range.
- `VM_SHARED` indicates that the pages are shared. Only one copy of these pages is kept in memory, rather than one copy per process. Otherwise, the pages are marked as private, and only the owning process can access them.
- `VM_RESERVED` indicates that the memory area cannot be swapped out. `VM_IO` indicates that this is a mapping of the processes I/O space. These are often used by device drivers.
- `VM_SEQ_READ` and `VM_RANDOM_READ` indicate to the kernel the types of accesses that are likely to occur (sequential or random). This allows the kernel to increase or decrease read-ahead for better performance.



- The address maps for a given process can be examined by looking at the output of `/proc/<pid>/maps` or by using **pmap**. An annotated output for a program (test) that just sleeps is shown below.
- The maps below correspond to the text (code), data (initialized variables) and bss (uninitialized variables) of the program (test), glibc (which the program was linked to), and the dynamic linker.
- The text memory areas have read and execute permissions, while the test and data segments are read-only or read-write. The bss segments are the anonymous segments that map to the device on inode 0. This corresponds to the zero page.
- Notice that 3M of memory is mapped, but only 128K is writable/private. The kernel will keep only one copy of memory areas which are not writable/private in memory, rather than one per process.
- The stack can be examined using **pstack**

```
$ pmap -d 2597
2597:  ./test
Address      Kbytes Mode  Offset                Device      Mapping
0000000000400000      4 r-x-- 0000000000000000 0fd:00002 test (text)
0000000000600000      4 rw--- 0000000000000000 0fd:00002 test (data)
0000003bda800000    108 r-x-- 0000000000000000 0fd:00000 ld-2.7.so (text)
0000003bdaa1a000      4 r---- 000000000001a000 0fd:00000 ld-2.7.so (data)
0000003bdaalb000      4 rw--- 000000000001b000 0fd:00000 ld-2.7.so (data)
0000003bdba00000   1332 r-x-- 0000000000000000 0fd:00000 libc-2.7.so ✓
(text)
0000003bdbb4d000    2048 ----- 000000000014d000 0fd:00000 libc-2.7.so (?)
0000003bdbbd4d000     16 r---- 000000000014d000 0fd:00000 libc-2.7.so ✓
(data)
0000003bdbd51000      4 rw--- 0000000000151000 0fd:00000 libc-2.7.so ✓
(data)
0000003bdbd52000     20 rw--- 0000003bdbd52000 000:00000 [ anon ] (bss)
00002aaaaaaab000      8 rw--- 00002aaaaaaab000 000:00000 [ anon ] (bss)
00002aaaaaaadd000      4 rw--- 00002aaaaaaadd000 000:00000 [ anon ] (bss)
00007fffc8c0c000     84 rw--- 00007fffffefa000 000:00000 [ stack ]
00007fffc8dfd000      8 r-x-- 00007fffc8dfd000 000:00000 [ anon ]
fffffffffff600000      4 r-x-- 0000000000000000 000:00000 [ anon ]
mapped: 3652K   writeable/private: 128K   shared: 0K
```

```
$ pstack 2597
#0 0x0000003bdba9ac30 in __nanosleep_nocancel () from /lib64/libc.so.6
#1 0x0000003bdba9aa84 in sleep () from /lib64/libc.so.6
#2 0x00000000004004d6 in main ()
```



- The kernel delegates some critical tasks to intermittently running kernel threads eg. flushing disk caches, servicing softirqs, flushing dirty buffers to disk
- These threads run in kernel space only
- They only use kernel space memory addresses (PAGE_OFFSET). The mm field in the process descriptor is NULL as there is no userspace component.
- Uses the `kernel_thread()` function to start the thread:

```
extern int kernel_thread(int (*fn)(void *), void * arg,  
unsigned long flags);
```

which calls `do_fork()` with the clone flags argument set to `flags | CLONE_VM | CLONE_UNTRACED`

`kernel_thread` (arch/i386/kernel/process.c) calls `do_fork()` as follows:

```
do_fork(flags | CLONE_VM | CLONE_UNTRACED, 0 , pregs, 0 , NULL, NULL)
```

where `pregs` is the memory address where the registers for `fn` and `arg` are stored. `do_fork()` will execute from `fn(arg)`.

Because the clone flags include `CLONE_VM`, the kernel thread will share its entire memory space - which only includes kernel space addresses - with other kernel threads and the kernel itself.





- also called the idle process or swapper process
- ancestor of all other processes
- created from scratch during linux initialization from statically allocated data structures
- initialized using the INIT_TASK macro in `include/linux/init_task.h`
- Process 0 will execute the `start_kernel()` function which creates process 1 (init)

```
kernel_thread(init, NULL, clone_FS^|CLONE_SIGHAND)
```




- Process 0 then calls `copy_process()` to make separate copies of the idle process on each CPU.
- Process 0 then calls `cpu_idle()`
- Process 0 is scheduled if nothing else is on the runqueue.
- Process 1 calls `execv()` and loads the init executable, which starts all other processes. A list of the running kernel threads can be obtained by doing a `ps` and looking for all the processes in [].

```
]$ ps -ef|grep "\["
root      1      0  0 Mar28 ?        00:00:02 init [5]
root      2      0  0 Mar28 ?        00:00:00 [kthreadd]
root      3      2  0 Mar28 ?        00:00:04 [migration/0]
root      4      2  0 Mar28 ?        00:00:08 [ksoftirqd/0]
root      5      2  0 Mar28 ?        00:00:00 [watchdog/0]
root      9      2  0 Mar28 ?        00:00:03 [events/0]
root     11      2  0 Mar28 ?        00:00:00 [khelper]
root     62      2  0 Mar28 ?        00:00:00 [kblockd/0]
root     66      2  0 Mar28 ?        00:00:00 [kacpid]
root     67      2  0 Mar28 ?        00:00:00 [kacpi_notify]
root    155      2  0 Mar28 ?        00:00:00 [cqueue/0]
root    158      2  0 Mar28 ?        00:00:00 [ksuspend_usbd]
root    163      2  0 Mar28 ?        00:00:00 [khubd]
root    166      2  0 Mar28 ?        00:00:00 [kseriod]
root    213      2  0 Mar28 ?        00:00:00 [pdflush]
root    215      2  0 Mar28 ?        00:00:00 [kswapd0]
root    259      2  0 Mar28 ?        00:00:00 [aio/0]
root    395      2  0 Mar28 ?        00:00:00 [pccardd]
root    409      2  0 Mar28 ?        00:00:00 [kpsmoused]
root    469      2  0 Mar28 ?        00:00:07 [ata/0]
root    471      2  0 Mar28 ?        00:00:00 [ata_aux]
root    475      2  0 Mar28 ?        00:00:00 [scsi_eh_0]
root    495      2  0 Mar28 ?        00:00:00 [ksnapd]
root    506      2  0 Mar28 ?        00:00:01 [kjournald]
root    534      2  0 Mar28 ?        00:00:00 [kauditd]
root   1306      2  0 Mar28 ?        00:00:00 [btaddconn]
root   1307      2  0 Mar28 ?        00:00:00 [btdelconn]
root   1501      2  0 Mar28 ?        00:00:00 [kmpathd/0]
```



- `exit()` C library call calls `exit_group()` system call, which calls the kernel function `do_group_exit()` This terminates a full thread group
- `pthread_exit` C library call calls `_exit()` system call, which calls `do_exit()`
- `do_group_exit()` calls `zap_other_threads()` which sends a `SIGKILL` to all other processes in the thread group. Each process then executes `do_exit()`.
- `do_exit()` does the following:
 - sets `PF_EXITING` flag in the process descriptor
 - removes process descriptor from any dynamic timers
 - detaches data structures for resources (`exit_mm`, `exit_sem`, `_exit_files`, `exit_fs` ..)
 - invokes the `exit_notify()` function to notify the parent
 - invokes the scheduler by calling `schedule()`



- Occurs when the scheduler suspends the operation of one process and resumes execution of another process
- Occurs only in kernel mode when the `schedule()` function is called:
- Calls `context_switch()` in `kernel/sched.c` in which:
 - kernel stores hardware context (cpu registers, kernel mode stack) in the `task_struct` thread field (type `thread_struct`)
 - installs the new address space by calling `switch_mm`
 - loads the new process hardware context onto the cpu registers
 - All inlined assembly code - must be very efficient



- Always occurs in kernel mode when the function `schedule()` is called
- `schedule()` will:
 - Check if the `current` task is still in the run queue
 - Check to see if the `need_resched` flag in the current `thread_info` struct is set
 - Find the highest priority runnable process (details in next module)
 - Check if this process is different from the current running process
 - If so, call `context_switch()` in `kernel/sched.c`

The `need_resched` flag used to be a single global variable per processor. However, because the `current` macro is so fast and likely to be in the cache line, the flag was moved to the `task_struct`.

In 2.2/ 2.4, the flag was an `int` in the `task_struct`. In 2.6, it is set to a field in the `thread_info` structure.



- `need_resched` is set using the macro `set_tsk_need_resched(struct task_struct *tsk)` in `sched.h` or by setting the flag directly

```
static inline void set_tsk_need_resched(struct task_struct *tsk)
{ set_tsk_thread_flag(tsk, TIF_NEED_RESCHED); }
```

- `need_resched` is set when:
 - A timer interrupt is serviced. The function `update_times()` updates the running time of the current process. It calls `scheduler_tick()` which sets `need_resched()` if current's timeslice is exhausted or if the process running is the idle process (process 0)
 - Kernel code blocks waiting for a resource (as in the page fault code)
 - A higher priority process wakes up and becomes active (in the `try_to_wake_up` function)
 - During the processing of a inter-processor interrupt.
 - A new process is forked



For a non-preemptive kernel:

- when returning from userspace or from servicing an interrupt or a system call
- only if `need_resched` is set
- safe to reschedule now - going back into userspace is a nice, quiescent state
- code is in `entry.S` (link)
- when the kernel explicitly blocks by calling `schedule` when resources are not available
- current task puts itself on a wait queue in `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` state (and off the run queue)

This system has two processes running P1 and P2.

1. `schedule()` is called when P1 is forked from P2. The scheduler inserts P1 before P2 in the active run queue at the same priority level.
2. P1 makes a system call and a mode switch occurs from user to kernel mode.
3. A hardware interrupt occurs. Kernel execution switches to the interrupt handler
4. A second (unmasked) interrupt occurs (perhaps on a different IRQ). Kernel execution switches to this interrupt handler.
5. A timer interrupt occurs and the timer interrupt handler code is executed. In `schedule_tick()` it is determined that the P1 has exhausted its timeslice. `need_resched` is set - but `schedule()` is not called. P1 is inserted in the expired array.
6. return from syscall. `schedule()` called. P2 is still in the active runqueue - so it is selected.
7. P2 makes a `read()` syscall. Mode switch occurs.
8. A page fault occurs. The pages are not available. P2 puts itself in `TASK_INTERRUPTIBLE` state and on a waitqueue by running `(fn)`. `schedule()` is called. No more tasks in the active queue - the expired queue becomes the active queue and P1 is selected.
9. A hardware interrupt occurs. The handler is called. This interrupt wakes up P2, which takes it off the waitqueue, recalculates the priority and inserts in the active queue. As its priority is greater than P1 (for example), `need_resched` is set.
10. Return from interrupt. `need_resched` is set, so `schedule()` is called and P2 is selected.



- From 2.6.18 kernel, preemption was added to the kernel.
- In the non-preemptive kernel, kernel code continues until it completes (returns to user space) or blocks.
- In the preemptive kernel, processes can be preempted even in kernel mode.
- current process can be preempted in the kernel if no locks are held
- field added to task_struct: preempt_count = number of locks held



End of Lecture 11

- Questions and Answers
- Summary
 - In this module, we have described what occurs when a process is created or destroyed. In particular, we have looked at the memory maps that are created for a process. We also looked at kernel threads and explained how the first process is started by the kernel on initialization.

We also discussed what happens during a context switch, and described when context switches occur in both a preemptive and non-preemptive kernel.





Lab 11.1: Kernel Threads

Scenario: In this lab exercise you will write a kernel module that identifies the tasks currently on the system which are kernel threads.

Deliverable: A kernel module which lists current processes which are kernel threads

Instructions:

1. Write a kernel module which identifies kernel threads currently in the system's task list. This should be done in the module's initialization function, then it should return with an error since it doesn't take any further action. This module should print the following information for each kernel thread it identifies:

- process ID
- process state
- command name

To save time, use your solution for lab 6 as a starting point for the code for this lab exercise.



Lab 11.2: Process Creation

Scenario: In this lab exercise you will write a program which uses the `clone()` system call to create another process.

Deliverable: A C program which creates another process which shares memory with its parent

Instructions:

1. Write a user-space application named `clone.c` which creates another process by calling the `clone()` library function. Create the new process such that it shares the same memory region as the original program, so any changes made to variables by either process are visible to both processes.

Write your program so you can clearly see that both programs share the same address space. Also use delays so you can monitor the processes while they execute using **top**, **ps**, and other system monitoring tools.

2. Modify your code so the child process calls `exit()` and terminates before the parent. What happens in this case?

parent who has exit transfer of memory

3. Modify your program so the parent process terminates before the child. What happens in this case?

parent have (child has memory, does)
(child) belongs to parent
ppid = 1 (myid process)

Lab 11.1 Solutions

1. Solutions for these exercises are found on the instructor machine at `/var/ftp/pub/rhd361-solutions/11-processes`. A Makefile is also provided that will compile the C source code appropriately.

Lab 11.2 Solutions

2. When the child process terminates before the parent, it becomes a zombie process until the parent calls `wait()`.
3. When the parent process terminates before the child, the child terminates immediately with it as well.







Lecture 12

The Scheduler

Upon completion of this unit, you should be able to:

- Describe how regular and real time processes are assigned priorities and time slices
- Describe the structures used by the $O(1)$ scheduler and explain how the $O(1)$ scheduler works
- Show some of the limitations of the $O(1)$ scheduler
- Explain how the new Completely Fair Scheduler works and is implemented in the upstream kernel



- Priority depends on the type of process:
 - Interactive processes - tend to be I/O bound
 - Batch processes - tend to be CPU intensive
 - Real-time processes - need a guaranteed short response time
- When `schedule()` kernel function completes, the highest priority process will be running on the CPU

The Linux scheduler has many competing demands. The goal is to allow all processes some access to the CPU, but preference should be given to higher priority, more critical, processes. One of the major factors that influences the scheduler is process type.

Interactive processes (like editors, graphical programs, etc.) interact constantly with users and spend most of their time waiting for key presses and mouse clicks. They must respond quickly and have a bounded variance in response time. Interactive processes are by nature I/O bound.

Batch processes (like compilers, mathematical computations, simulations, etc.) can execute in the background. They do not need to be responsive, but require a lot of CPU time to complete. By nature, batch processes are CPU bound.

Finally, real-time processes need a guaranteed short response time with low variance. Examples of these types of processes include robot controllers and stock trading applications.



- Effective priority is dynamically calculated by the scheduler and stored in the `task_struct` field `prio`
- Value ranges from 100 to 140 (lower value = higher priority)
- Priority is calculated based on a static priority determined by the niceness value and a dynamic priority calculated by the scheduler.
- Nice-ness values range from -20 to +20 (default 0) with -20 (least nice) being highest priority. Niceness can be read/set using the system calls `getpriority()`, `setpriority()`
- Interactive processes are identified heuristically by the scheduler as those with large sleep average. They are given a bonus to boost priority. Batch processes are penalized.

The formula for calculating priority as seen in `recalc_prio()` is roughly as follows:

```
prio = 120 + nice - bonus
```

with `bonus` ranging from +5 (most interactive) to -5 (least interactive), and capping the priority within the 100-140 range. The bonus calculation has been heuristically determined and is pretty complicated, but works quite well.



- Always higher than normal processes
- Ranges from 1 - 99
- Set by the `sched_setscheduler()` system call
- Scheduling policy can be one of three types
 - `SCHED_FIFO`
 - `SCHED_RR`
 - `SCHED_OTHER`

SCHED_FIFO: There is no concept of time slices. The process is scheduled until it either blocks and becomes non-runnable or yields by calling a `sched_yield`. When it returns to userspace, unless a higher priority task has become active, it will continue to run, even if there are other runnable processes of the same priority.

SCHED_RR: All tasks of a given realtime priority will be allocated a fixed timeslice. When one process has completed its time slice, a second (equal priority) process will run (in a round robin fashion).

SCHED_OTHER: Used for normal processes. Same as described before.



- Time slices for conventional processes are determined as a function of the priority.
- When processes are first forked, they split the parent's remaining time slice between them (to avoid one process forking new children to get an unlimited timeslice)
- Subsequent time slices are calculated based on the static priority in the function `task_timeslice()`
- For `SCHED_RR`, the time slice can be obtained by calling `sched_rr_get_interval`

Time slices are calculated based on the following formula:

If priority $p < 120$:

$$\text{timeslice (in ms)} = \max (20 * (140 - p), 5\text{ms})$$

If priority $p > 120$:

$$\text{timeslice (in ms)} = \max (5 * (140 - p), 5 \text{ ms})$$

The above formulas assume the HZ value is set to 1000. Based on the above formula, a process with the minimum priority (maximum nice value of +19) would receive the minimum timeslice of 5ms. A process with a default priority (nice value of 0) would receive a 100ms timeslice, and a process with a nice value of -20 would receive a 800ms timeslice.



- The O(1) scheduler was introduced in the 2.6 kernel
- Each processor has a single runqueue as defined in `kernel/sched.c`
 - Macros for accessing runqueues: `this_rq()`, `cpu_rq(cpu)`, `task_rq(task)`
- Each runnable process is in only one runqueue
- Runqueues are protected by spinlocks during modification
 - Macros to lock and unlock are: `task_rq_lock()`, `task_rq_unlock()`, `this_rq_lock()`, `this_rq_unlock()`

The runqueue (defined in `kernel/sched.c`) contains many fields. The most important ones are:

```
struct rq {
    spinlock_t lock;
    unsigned long nr_running;
    unsigned long raw_weighted_load;
#ifdef CONFIG_SMP
    unsigned long cpu_load[3];
#endif
    unsigned long long nr_switches;
    unsigned long nr_uninterruptible;
    unsigned long expired_timestamp;
    unsigned long long timestamp_last_tick;
    struct task_struct *curr, *idle;
    struct mm_struct *prev_mm;
    struct prio_array *active, *expired, arrays[2];
    int best_expired_prio;
    atomic_t nr_iowait;

#ifdef CONFIG_SMP
    struct sched_domain *sd;

    /* For active balancing */
    int active_balance;
    int push_cpu;
    int cpu; /* cpu of this runqueue */

    struct task_struct *migration_thread;
    struct list_head migration_queue;
#endif
    struct lock_class_key rq_lock_key;
};

static DEFINE_PER_CPU(struct rq, runqueues);
static inline int cpu_of(struct rq *rq)
{
#ifdef CONFIG_SMP
    return rq->cpu;
#else
    return 0;
#endif
}
```

```
}
```

Note that the struct contains a reference to the current task_struct and two priority arrays (active and expired).



- Each runqueue contains two priority arrays.
- A priority array (defined below) contains:
 - an array of 140 linked lists consisting of all the tasks with that priority. So queue[5] is a linked list of all the tasks with priority 5.
 - a bitmask with enough bits to denote all 140 priority levels. (5 bits) the left-most bit that is set in this array will denote the highest priority process that is active and runnable.

Definition of a prio_array (kernel/sched.c)

```
192 struct prio_array {  
193     unsigned int nr_active;  
194     DECLARE_BITMAP(bitmap, MAX_PRIO+1); /* include 1 bit for  
delimiter */  
195     struct list_head queue[MAX_PRIO];  
196};
```

where MAX_PRIO is the highest priority number (which is 140).



- When a task becomes runnable, its priority is calculated and it is placed in the corresponding linked list in the active priority array, and sets the appropriate bit in the bitmask.
- When `schedule()` is called, the scheduler looks for the left-most bit that is set in the bitmask. (`sched_find_first_set_bit`) This is the priority of the highest priority active process to be run. The scheduler then selects a process from that queue to run.
- When a process uses up its timeslice, it has its priority recalculated and is placed in the expired runqueue. Expired processes do not run until all processes in the active array have run.



- When all active processes are run, the active and expired arrays are switched. This is as simple as swapping the pointers to the arrays
- Interactive processes are allowed to remain in the active priority array even after having used up their timeslices, unless some processes in the expired array have been starved of processor time for too long
- Finding the highest priority task is as simple as a lookup. It does not depend on the number of processes in the system. In fact, as the number of priorities available is fixed, the algorithm completes in constant time. Hence the name O(1) scheduler.
- This is much better than the 2.4 scheduler which had to scan through all the tasks in the system to find the highest priority task. This scaled as O(n) where n is the number of tasks on the system.

Tasks have their time slice incremented in the `timer_tick()` function. In this function, if the process time slice has been exhausted, the process is enqueued on the expired array if it is not interactive, according to the code below:

```
struct task_struct *task = current;
struct runqueue *rq = this_rq();

if (!--task->time_slice) {
    if (!TASK_INTERACTIVE(task) || EXPIRED_STARVING(rq))
        enqueue_task(task, rq->expired);
    else
        enqueue_task(task, rq->active);
}
```

This checks to see if any expired processes are starving first. If so, the (current) interactive task is enqueued on the expired array instead.



- When processes are blocked or sleeping - by explicitly sleeping, waiting for I/O or waiting for a contested semaphore, the task is set to either `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` and placed on a wait queue.
- Wait queues are represented in the kernel by `wait_queue_head_t` which are declared statically by `DECLARE_WAITQUEUE()` or dynamically by `init_waitqueue_head()`
- The code below from `sched.c` shows how the current task can wait for completion of an event `x` by creating a wait queue entry by calling `DECLARE_WAITQUEUE` and adding the entry to the end of the wait queue `&x->wait`.

If the condition `x->done` is false and no signals are pending, then the task state is set to `TASK_INTERRUPTIBLE`, spin locks are released, and `schedule()` is called.

- At some point, the event `x->done` becomes true and the driver calls `wake_up` to wake all the tasks waiting on the queue `&x->wait`. The task state is set to `TASK_RUNNING` and it is placed in the runqueue. When it does run, it checks the condition again (for spurious wake ups) and removes its entry from the wait queue.

```
int fastcall __sched wait_for_completion_interruptible(struct completion
*x)
{
    int ret = 0;

    might_sleep();

    spin_lock_irq(&x->wait.lock);
    if (!x->done) {
        DECLARE_WAITQUEUE(wait, current);

        wait.flags |= WQ_FLAG_EXCLUSIVE;
        __add_wait_queue_tail(&x->wait, &wait);
        do {
            if (signal_pending(current)) {
                ret = -ERESTARTSYS;
                __remove_wait_queue(&x->wait, &wait);
                goto out;
            }
            __set_current_state(TASK_INTERRUPTIBLE);
            spin_unlock_irq(&x->wait.lock);
            schedule();
            spin_lock_irq(&x->wait.lock);
        } while (!x->done);
        __remove_wait_queue(&x->wait, &wait);
    }
    x->done--;
out:
    spin_unlock_irq(&x->wait.lock);

    return ret;
}
```




- Each CPU has its own independent runqueue with its own list of tasks and scheduler. On an SMP system, processors could become unbalanced - with many more tasks on one processor than another.
- To prevent imbalance, the scheduler calls the function `load_balance()` to pull tasks from other CPUs:
 - in `schedule()` when the runqueue is empty
 - during a timer tick, every 1 ms if the system is idle or every 200 ms otherwise



- The `load_balance()` function does the following:
 - Locks the runqueue and disables interrupts.
 - Calls `find_busiest_queue()` to determine the runqueue with the greatest number of processes on it. If the runqueue has at least 25% more processes than the current, this queue is returned.
 - Selects the expired array if possible. These processes have probably not run in awhile and are not cache hot.
 - Selects the highest priority process that is not running, not prevented from migration through processor affinity and not cache hot, and calls `pull_task()`
 - Repeats until the imbalance is resolved.



- Timeslice (share of the CPU) is difficult to calculate
- Interactive tasks are treated special via heuristics
- In-kernel auto-renicing to avoid starvation of higher nice levels
- Fairness is difficult to achieve



- O(1) Scheduler
 - Singular priority array for both RT and non-RT tasks
 - Single decision algorithm based upon priority array position
 - Fairness is calculated by complex heuristics
- Completely Fair Scheduler (CFS)
 - Introduced in kernel 2.6.23
 - Separate queues mechanisms for RT and non-RT tasks
 - Separate decision algorithms based on scheduler classes
 - Fairness is calculated using pure math



- CONFIG_FAIR_GROUP_SCHED
- Complete re-write of the scheduler:
 - No runqueues
 - Nanosecond time resolution
 - No timeslices and does not rely upon jiffies
 - No heuristics
 - Computing "fairness" is a matter of simple math
 - Extensible framework
 - Simple to introduce new scheduler algorithms
 - Pluggable scheduler
- Based on the idea that every process should get its fair share of the CPU



- Models "ideal, precise multi-tasking CPU" on real hardware
 - Ideally, all runnable processes would run simultaneously, each using up an equal fraction of the CPU
- On a real machine, only one process can run at any time. This means that during the time that this process runs, it is using up more than its fair share of the CPU (100%), while other waiting process are at a disadvantage.
- This unfairness is tracked in a per-process variable added to the task_struct p->wait_runtime
- wait_runtime is the amount of time a process would need to run on the CPU for it to be completely fair and balanced. On perfect hardware, this value would be zero.
- CFS would select the task which has the highest wait_runtime



- Tasks are ordered in a red-black tree, based on the value of `wait_runtime`.
- Picking the task to run is straightforward and is $O(1)$. The task to run is just the leftmost node in the tree.
- When `schedule()` is called, the current task's `wait_runtime` is reduced by the amount of CPU time it has just expended (minus its fair share).
- The task is then reinserted in the red-black tree. This process is $O(\log n)$. The semantics of a red-black tree ensure that the tree is never too deep.
- If the process is no longer the left most node (minus a little granularity in distance to the leftmost node to avoid trashing the cache), then the process is preempted and a context switch occurs.
- There is only one tunable - `/proc/sys/kernel/sched_granularity_ns` which can be tuned for low latencies (interactive loads) or batch loads.



- `SCHED_NORMAL`
 - Policy used for "regular" tasks
- `SCHED_BATCH`
 - Preemption minimized to allow tasks to run longer and make better use of caches
 - Interactive jobs are not well suited for this
- `SCHED_IDLE`
 - Less priority than nice 19
 - Not a true idle timer to avoid priority inversion problems
- `SCHED_FIFO` and `SCHED_RR`
 - POSIX realtime policies
 - `kernel/sched_rt.c`

The command **chrt** can view and set all scheduling policies except `SCHED_IDLE`.

```
# chrt -p 1
pid 1's current scheduling policy: SCHED_OTHER
pid 1's current scheduling priority: 0
# ps axo pid,comm,rtprio,policy | grep Xorg
5477 Xorg                - TS
```



- Extensible, hierarchical set of classes implemented as `struct`
- Scheduler decision looks for first runnable task in the top scheduler, cascading through successive scheduler classes until it finds one
- `rt_sched_class`
 - Manages `SCHED_FIFO` and `SCHED_RR` tasks
 - $O(1)$ priority array
- `fair_sched_class`
 - Manages `SCHED_OTHER` tasks
 - $O(\log(N))$ red-black tree
- `idle_sched_class`
 - For the idle task

CFS makes a scheduler decision based upon a simple algorithm. CFS looks first at the top scheduler class to determine if there is a runnable task available. If so, the next task is taken off the list and scales as $O(1)$. If not, CFS looks at the next scheduler class for the first runnable task available.



- Nanosecond-based accounting
 - Independent of HZ and jiffies
 - High-Resolution Timers are used, if available
 - No notion of previous scheduler's timeslices
 - No heuristics
- Uses a red-black tree to manage task list
 - organizes tasks as a timeline of tasks to schedule
- Determining the next task to schedule is $O(1)$
 - The task with longest wait time in the red-black tree
- Inserting a new task into the tree is $O(\log(N))$
- CFS nice levels:
 - $\text{weight}(n) = 1.25 * \text{weight}(n-1)$
 - Don't depend upon the timeslice
 - Multiplicative
- Interactive tasks are given a priority boost
- Sleep time is tracked

For more information about how CFS implements nice values, see <http://kerneltrap.org/node/11773>.

Scheduling algorithms provide bonuses to interactive tasks. Sleeping tasks get a boosted priority, but have a priority "ceiling" that is set to the lowest priority that would still allow it to be reinserted back into the active scheduling array upon "timeslice completion". The ceiling prevents a task from achieving a best-priority value by using a long-enough single sleep.



- CFS can be tuned:
 - "Desktop" (low latencies)
 - "Server" (batch workloads)
- CONFIG_SCHED_DEBUG required
- Tunable:
 - `/proc/sys/kernel/sched_min_granularity_ns`
- Defaults to setting for typical desktop workloads



- Available in 2.6.24
- Tasks can be grouped by (one or the other, only):
 - UID (`CONFIG_FAIR_GROUP_SCHED`)
 - User-defined control groups (`CONFIG_FAIR_CGROUP_SCHED`)
- Example of group scheduling:
 - 100 tasks scheduled
 - 1 is Joe's server, 99 are Jane's
 - Joe's server gets 50% of the CPU, and Jane's 99 tasks share the other 50% of the CPU
 - Configurable: 75% Joe, 25% Jane for example

Normally, the scheduler looks at each task individually and independently to determine, as fair as it can, an appropriate amount of CPU time to assign each. Consider the case of Joe running a high-priority service and Jane running many tasks as part of her lower-priority compilations. The magnitude of Jane's tasks may unfairly eat away at the amount of CPU time that ought to be allotted to Joe's service.

CFS group scheduling allows two forms of grouping to improve the mechanism. Grouping by UID, and grouping by arbitrarily-named groups. In the case of Joe and Jane, grouping by UID the scheduler can achieve a much fairer balance (or, in fact, administrator configurable preference of priority balances) by simply accumulating all of Jane's tasks into a single scheduler entity which is balanced with Joe's scheduling entity. Joe's task might get 50% of the CPU and all of Jane's tasks might share a 50% allotment, or the system administrator can configure Joe's UID to be granted a much higher share of the CPU than all other UIDs.



- CPU time is shared according to UID
- A directory is created in sysfs for each new user containing a file with that user's *CPU share*
 - `/sys/kernel/uids/UID/cpu_share`
- CPU bandwidth is divided according to `cpu_share` values
- Example (UID 502 gets twice the CPU share of UID 501):

```
# cd /sys/kernel/uids
# echo 1024 > 501/cpu_share
# echo 2048 > 502/cpu_share
```

Beginning with the scheduler in 2.6.24, modifying scheduler priorities on a per-user basis gets a lot easier. A subdirectory named after each user's UID on the system will be automatically created in the sysfs directory `/sys/kernel/uids` containing a file named `cpu_share`. A default integer value of 1024 will be inserted into each user's `cpu_share` file. The scheduler will adjust priorities according to the `cpu_share` value ratio between two or more users in the active queue.



- CPU time is shared according to arbitrary group names
- Group scheduling implemented using *process container* mechanism:
 - Admin mounts a container filesystem with `cgroup` type and `cpu` option
 - Scheduling groups are created as directories in the filesystem
 - Processes can be moved in and out of the groups
 - Allows for arbitrary groups (e.g. compiling, browser)
- Custom policies can be implemented via user-space daemons responding to the groups
- A `cpu.shares` file is created for each group to modify their CPU share
- Example (browser gets twice the CPU share of a compilation):

```
# mkdir /dev/cgroup
# mount -t cgroup -o cpu none /dev/cgroup
# cd /dev/cgroup
# mkdir {compiling,browser}
# echo 1024 > compiling/cpu.shares
# echo 2048 > browser/cpu.shares
# make &
# pidof make > compiling/tasks
# firefox &
# pidof firefox > browser/tasks
```

See the files in `Documentation/scheduler` (especially `sched-design-CFS.txt`) for more information about CFS and grouping.



End of Lecture 12

- Questions and Answers
- Summary
 - Examined how Linux determines which runnable process should be executed based on the process priorities
 - $O(1)$ scheduler implementation and limitations
 - Completely Fair Scheduler (CFS) implementation



Lab 12.1: Scheduling Standard Priority Processes

Scenario: In this lab you will write a CPU intensive program which occasionally produces output so you can observe how it is assigned CPU resources by the Linux kernel.

Deliverable: A program which can have its standard process priority changed when invoked

Instructions:

1. Write a C program called `dotter-nice.c` which repeatedly executes a tight loop up to a fixed count, then prints a period to standard error each time the loop completes. Adjust the fixed constant used for the tight loop so that a full line of 80 periods takes between one second and five seconds to print under normal system load.

Note: The output should go to standard error instead of standard output because standard output is line buffered when outputting to the display.

Write your program so it takes an optional argument which is a valid nice value or priority it should run as. When the argument is omitted the program should run with a default process priority.

2. Execute your program under different system loads. A simple way to load the CPU is to execute the following command in the background:

```
[root@host ~]# cat /dev/zero > /dev/null &
```

3. Observe how the scheduler runs your program on an unloaded system versus when there are multiple background processes running which are CPU intensive.

It is easiest to see how the scheduler works on a uniprocessor system. On a multiprocessor machine run the **multilock** program with a single argument equal to the number of CPUs on your system minus 1. It will keep them busy.

When the system is loaded, what is the behavior of your program when it initially starts executing? Can you explain why it behaves this way?

4. Run your program with different priorities. Can you see a difference in how it gets scheduled and can you explain why?
-

Lab 12.2: Scheduling Real-Time Priority Processes

Scenario: You will modify the program from the previous lab exercise to adjust its scheduling policy and priority to be a real-time process. This will allow you to observe how the Linux scheduler handles real-time processes.

Deliverable: A program which can be assigned a real-time process priority with either `SCHED_RR` or `SCHED_FIFO` scheduling policy when invoked

Instructions:

1. Modify a working program from the previous lab exercise and call it `dotter-rt.c`. It should require two command-line arguments. The first argument is a single letter, "F" or "R", indicating the scheduling policy this process should use, `SCHED_FIFO` and `SCHED_RR` respectively. The second argument is an integer between `sched_get_priority_min()` and `sched_get_priority_max()` for the selected policy indicating the process' real-time priority.

Experiment with your real-time priority program and see how the Linux scheduler works with it.

Lab 12.1 Solutions

1. Solutions for these exercises are found on the instructor machine at `/var/ftp/pub/rhd361-solutions/12-scheduler`. A `Makefile` is also provided that will compile the C source code appropriately.
3. Observe how the scheduler runs your program on an unloaded system versus when there are multiple background processes running which are CPU intensive. What is the behavior of your program when it initially starts executing?

Your program should operate at full speed for almost a second then slow down dramatically. Initially it runs at a higher priority than the other CPU-bound processes when it is first spawned by the shell. When the scheduler detects it is a CPU-bound process as well, it lowers its priority to be equal with the other CPU-bound processes so it must compete for the CPU.

Lab 12.2 Solutions

1. When you test your real-time priority program, you may find the system hangs and becomes unusable. How can you get control back? Is the Magic SysRq key enabled?

Your CPU-intensive will run at a higher priority than the rest of the interactive programs on your system. To see the output of your program you will have to increase the priority of the shell running it so the shell can occasionally run and display the output to the screen.

Things get more complicated if you are using X, so you will have to be clever to see your program's output. Use **ssh** to access your system, then use the **chrt** command to assign your shell and the underlying **sshd** process a real-time priority higher than your program.

Note: Be careful not to launch other programs from your elevated priority shell since they will inherit the shell's priority and influence the results of your tests.





Lecture 13

Kernel Timing

Upon completion of this unit, you should be able to:

- Explain the purpose of kernel timing
- Query wall time in the kernel
- Use appropriate macros for `jiffies` comparisons
- Schedule kernel functions for deferred execution
- Manipulate kernel high-resolution timers



- Occasional ticks needed for multiprocessing
- Allows user processes to set alarms and sleep
- Keep track of wall time (time stamps)
- Delays and timeouts for I/O devices

Linux is not a real time operating system in the sense that tasks can be guaranteed to run within a specified time period. The way the Linux timing architecture is defined, deferred tasks will run after a delay or a specified deadline has been reached.



- Real Time Clock (RTC)
- Local CPU APIC Timer
- Programmable Interval Timer (PIT)
- Time Stamp Counter (TSC)
- ACPI Power Management Timer (PMTMR)
- High Precision Event Timer (HPET)

The Real Time Clock (RTC) is a battery backed up time source that keeps track of the number of seconds since the UNIX Epoch, midnight GMT on January 1, 1970. It is the time source used to initialize system time when the system boots.

Modern processors have a local advanced programmable interrupt controller (APIC) which includes a local CPU timer. This timer is used to keep track of process time on the local CPU and raise interrupts local to that processor on a multiprocessor system.

The programmable interval timer (PIT) is a time that can be set to issue interrupts to the kernel at a specified interval. The PIT is often used for all general timekeeping activities, including process scheduling.

The time stamp counter (TSC) is a register within the CPU that is incremented on each pulse of the oscillator used to drive the CPU. It is a very high resolution time source, but sometimes can be inaccurate when a processor is throttled back to save power.

ACPI Power Management Timer (PMTMR) is another possible clock source that is available on some southbridges.

High Precision Event Timer (HPET) was designed for multimedia applications. It is a very precise and flexible time source that can be programmed to have multiple alarms of varying frequency.



- **Timer operations:** `struct timer_opts`
 - `mark_offset` - function called by timer interrupt
 - `get_offset` - returns microseconds since last timer interrupt
 - `monotonic_clock` - returns nanoseconds since clock initialization
 - `delay` - function that delays a number of clock cycles
- `cur_timer` points to time source being used
 - Initialized at boot time by `select_timer()`

The `select_timer()` function is used at boot time to initialize the time source to be used by the kernel. It returns the address of the most reliable source and it is stored in the `cur_timer` global variable. It tries to select the following time sources in order from most accurate to least:

- High precision event timer (HPET)
- ACPI power management timer
- Time stamp counter (TSC)
- Programmable interval timer (PIT)

All the fields of the `timer_opts` structure are pointers to functions except the `name` field which is a string that contains the name of the timer.



- `struct timespec xtime`
 - Contains seconds and nanoseconds since beginning of the UNIX Epoch
 - Protected with a read/write sequence lock: `xtime_lock`
- Initial value comes from hardware RTC
 - Later updated with timer hardware interrupts
- Value of RTC can be modified with `hwclock` command
 - This utility can also be used to modify `xtime` as well

Since the kernel's internal wall clock `xtime` is a `timespec` structure, it cannot be read or updated in an atomic manner. When nanosecond resolution is needed, best practice is to use the `current_kernel_time()` function to read `xtime` since it secures the `xtime_lock` sequence read/write lock used to protect `xtime` appropriately. Note securing the `xtime_lock` lock isn't necessary when reading only the `tv_sec` (seconds) field of `xtime`.

Another `timespec` structure global variable is `wall_to_monotonic`. It is a negative offset value which yields the system uptime when added to `xtime`.

File systems can utilize timestamps that are lower than nanosecond resolution. The `current_fs_time()` function returns the current time truncated to the resolution of the file system whose superblock is passed as a parameter. It uses the more general purpose `timespec_trunc()` function to perform the actual truncation as seen below:

```
struct timespec current_fs_time(struct super_block *sb)
{
    struct timespec now = current_kernel_time();
    return timespec_trunc(now, sb->s_time_gran);
}
```

The `{get,set}timeofday()` system calls allow userspace programs to manipulate the system clock using a `timeval` structure which is accurate within microsecond resolution. The `do_{get,set}timeofday()` functions are used to implement these two system calls once below the system call interface. They are exported symbols which can be used by kernel modules. Both system calls also refer to a `timezone` structure which is maintained internally in the kernel as the `sys_tz` global variable.



- `gettimeofday()/settimeofday()` functions
 - Passed a pointer to `timeval_t` structure
 - `timezone` structure is ignored
- `time()` function
 - Returns current time with second resolution
- All functions access `xtime` structure

The kernel functions that do most of the work reading and modifying the real time/wall clock are `do_gettimeofday()` and `do_settimeofday()`. These functions read and update the `xtime` global variable in the kernel that contains the current time at nanosecond resolution. The following code from `kernel/time.c` demonstrates how `do_gettimeofday()` uses `xtime` to determine the current time:

```
void do_gettimeofday (struct timeval *tv)
{
    unsigned long seq, nsec, usec, sec, offset;
    do {
        seq = read_seqbegin(&xtime_lock);
        offset = time_interpolator_get_offset();
        sec = xtime.tv_sec;
        nsec = xtime.tv_nsec;
    } while (unlikely(read_seqretry(&xtime_lock, seq)));

    usec = (nsec + offset) / 1000;

    while (unlikely(usec >= USEC_PER_SEC)) {
        usec -= USEC_PER_SEC;
        ++sec;
    }

    tv->tv_sec = sec;
    tv->tv_usec = usec;
}
```




- `jiffies` global variable of type `unsigned long`
 - Default resolution is 1 millisecond
 - Used by scheduler and default timer subsystem
- `HZ` and `SHIFT_HZ` macros
- `jiffies_64`
 - Access is not atomic on 32-bit architectures
 - Best to get its value with `get_jiffies_64()`
- `tick_divider` kernel parameter

It is likely the internal `jiffies` counter will roll back to zero on a 32-bit machine. It is initialized to -5 minutes at boot time so it will rollover soon after the kernel initializes itself. This is done to detect errant kernel code that uses `jiffies` inappropriately. Macros should be used to perform comparisons between two `jiffies`-based timer values:

```
#include <linux/jiffies.h>
time_after()
time_after_eq()
time_before()
time_before_eq()
```

The `HZ` macro is the interrupt timer frequency. It is initialized by a compile-time option, `CONFIG_HZ`, which is set to 1000 on normal kernels and 250 on Xen kernels. The `SHIFT_HZ` macro is power of 2 closest to `HZ`. `SHIFT_HZ` is used when a close approximate to `HZ` needs to be made without invoking a multiply instruction.

The `tick_divider` global variable is used to reduce the frequency of the timer interrupt. By default the interrupt occurs `HZ` times per second. When `tick_divider` is set on the kernel command line, the timer interrupt frequency is divided by that amount. For example, if `tick_divider` is specified with a value of 4, the timer interrupt frequency will be `HZ/4` or 250 interrupts per second. Another macro is defined, `REAL_HZ`, which represents the value of `HZ/tick_divider`.



- Structure `timer_list` fields
 - `function` is the deferred code to execute passing data as a parameter
 - `expires` is the absolute time in jiffies when timer is to fire
- Efficient processing of timers requires some consolidation
 - `round_jiffies()/round_jiffies_relative()` functions
- Putting a process to sleep for a period of time
 - `schedule_timeout_interruptible()` and its uninterruptible counterpart
 - Both functions change the state of the current process and set an alarm

The `timer_list` structure requires proper initialization before a timer can be scheduled to execute. Best practice is to zero the structure then pass its address to `init_timer()` for further initialization. Next programmers must assign the `function` and `data` fields to point to their specific function. The value of the `data` field, usually a pointer, is passed as the only parameter to the function specified in the `function` field. A macro called `setup_timer()` can be used to accomplish the above individual steps.

Timers are scheduled to execute when they are modified using the `mod_timer()` function. This function assigns the `expires` field with the absolute time in jiffies specified as an argument and places the timer in the appropriate place in the pending timer vectors. The `add_timer()` inline function simply calls `mod_timer()` to schedule the timer and generates an error if the timer has already been scheduled. To cancel a pending timer the `del_timer()` and `del_timer_sync()` functions should be used. `del_timer_sync()` deletes the timer and waits for the function to finish if it is executing so it should not be called from within an interrupt context. A good example of timer initialization and use can be found in `Documentation/connector/cn_test.c`.

For CPU performance reasons it is sometimes advantageous to consolidate some timer functions to execute together at the top of the second. The `round_jiffies()` function is used to round an absolute timer expiration to within the nearest second. A related function, `round_jiffies_relative()`, performs the same action but with a jiffies value relative to the current time.

Sometimes a programmer wants to put a process to sleep for a period of time. The `schedule_timeout_interruptible()` and `schedule_timeout_uninterruptible()` functions both use `schedule_timeout()` to use a timer as an alarm clock for a process. First each function sets the current process' task state to `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` respectively. Then they set a timer to go off a set number of jiffies in the future which will wake up the current process. Finally, the functions call `schedule()` to relinquish the processor to a runnable process. The `msleep()` and `msleep_interruptible()` functions are wrappers for the above `schedule_timeout` based functions where the sleep time is specified in millisecond resolution.



- Microsecond-resolution one-shot timers
 - Generate a `SIGALRM` signal when they expire
- Operate in four time domains in Linux
 - Real time: `CLOCK_REALTIME`, `CLOCK_MONOTONIC`
 - CPU time: `CLOCK_PROCESS_CPUTIME_ID`, `CLOCK_THREAD_CPUTIME_ID`
- Primary data types: `timer_t` and `itimerspec` structures
 - `timer_create()/timer_delete()`
 - `timer_settime()/timer_gettime()`
 - `timer_getoverrun()`

POSIX timers provide the capability for high-resolution one-shot timer programming. They have nanosecond resolution because their primary time measurement data type, the `itimerspec` structure, contains two `timeval` structures within it: `it_interval` and `it_value`. Only the `it_value` structure is used.

The code fragment below shows how to create a POSIX timer based on real time with an expiration of .35 seconds:

```
#include <time.h>
#include <signal.h>
...
timer_t my_timer;
structure itimerspec itspec;
...
timer_create(CLOCK_REALTIME, NULL, &my_timer);
itspec.it_value.tv_sec = 0;
itspec.it_value.tv_nsec = 350000000;
timer_settime(my_timer, 0, &itspec, NULL);
```

The `NULL` parameter to `timer_create` indicates that default the default signal `SIGALRM` should be sent to the process when the timer expires. At this point in the code it isn't programmed to go off yet. The call to `timer_settime()` programs the timer to fire according to the value in `itspec`.

The `timer_gettime()` function can be used to query a process' timer to see what remaining time is left. The `timer_delete()` function can remove the timer entirely.

When a timer expires, a signal normally gets sent to the process which created it. If a signal of the same type is pending when a timer expires, that is considered to be a timer overrun and a counter is incremented each time another timer expires before the signal is actually handled. The `timer_getoverrun()` function returns the number of overruns that have occurred for the specified timer.



- These functions provide userspace interface to timers
- alarm() generates SIGALRM when it expires
 - Low second-resolution only
- setitimer()/getitimer() provide microsecond resolution
 - Three different time domains in Linux
 - ITIMER_REAL: timer based on real time
 - ITIMER_VIRTUAL: timer updated only when process executes
 - ITIMER_PROF: ITIMER_VIRTUAL + system time spent
 - Generate different signals based on the time domain
 - Can be programmed to be recurrent

Interval timers generate different signals to the userspace application when they expire depending on which time domain they are in:

- ITIMER_REAL: generates SIGALRM upon expiration
- ITIMER_VIRTUAL: generates SIGVTALRM
- ITIMER_PROF: generates SIGPROF

The following code fragment shows how to program an interval timer to go off 5 seconds in the future and then every 1.5 seconds from then on:

```
#include <sys/time.h>
...
struct itimerval itv;
...
itv.it_value.tv_sec = 5;           # initial timer offset from now
itv.it_value.tv_usec = 0;
itv.it_interval.tv_sec = 1;       # interval thereafter
itv.it_interval.tv_usec = 500000;
setitimer(ITIMER_REAL, &itv, NULL);
```

Note the program must handle the SIGALRM signal otherwise it will terminate the first time the alarm goes off.



- `hrtimers` are 64-bit timers
 - Ideally they have nanosecond resolution
 - Will round down to a lower resolution clock if necessary
 - `ktimer_t` is the basic time data type
- Uses a red-black tree to organize timers
- Used to implement interval timers and POSIX timers

High resolution timers (`hrtimers`) were introduced in the 2.6.16 kernel. Time is represented as a `ktimer_t` type which is a 64-bit timer with nanosecond resolution. These timers will never overflow and wrap back to zero since the 64-bit value can represent over 540 years of nanosecond ticks.

```
struct hrtimer {
    ktimer_t expires;
    int      (*function)(struct *hrtimer);
    ...
}
```

The handlers used with `hrtimers` are passed the address of the `hrtimer` structure which is usually a field in a more general data structure defined as a `struct`. The handler can use the `container_of()` macro to get access to the data surrounding the embedded `hrtimer` structure. After the work is performed, it either returns `HRTIMER_NORESTART` or it schedules itself to run again using the `hrtimer_forward()` function and returns `HRTIMER_RESTART`.



- `timer_interrupt()` function
 - Defined in `arch/i386/kernel/time.c`
- Tasks performed by interrupt handler include:
 - Acknowledge the IRQ
 - Increment `jiffies_64`
 - Update `xtime`
 - Calculate and update load average
 - Update process user or system time
 - Deduct a tick from current process' scheduled quantum
 - Add expired POSIX thread and process timers to a "fired" list

The `timer_interrupt()` function interacts with the architecture specific hardware - mostly acknowledging the timer interrupt. Most of the real work is performed by the architecture independent functions `do_timer()` and `update_process_times()` (both defined in `kernel/timer.c`).

The `do_timer()` function is responsible for updating the system clocks. It increments `jiffies_64`, updates the kernel's real time variable: `xtime`, and updates the load average (maintained in an array named `avenrun`). All of these updates have a system-wide impact.

The `update_process_time()` function does time related tasks that pertain to the current process and processor. This function updates the current process' user or system time by one tick depending on what mode it was executing in. It calls the `run_local_timers()` function which performs the following tasks:

- Schedules `TIMER_SOFTIRQ` tasks to run on the current processor
- Reclaims old/unused RCU data structures
- Deducts a tick from the current process' quanta and possibly reschedules
- Moves expired process or POSIX thread timers to the fired queue



- `run_timer_softirq()` function
 - Defined in `kernel/timer.c`
- Tasks performed by the softirq include:
 - Process expired high resolution timers
 - Timers with deadlines based on absolute time
 - Timers based on system uptime
 - Process standard jiffies-based timers
 - Runs all expired timers scheduled on the current CPU
 - Enables interrupts during execution of each timer function

When the system-wide lists of timers are referenced or modified, a spinlock is set and interrupts on the current CPU are disabled. When each timer function is executed, the spinlock is unlocked and interrupts are enabled since each timer function could take a substantial amount of time to execute. This is the case for both the functions that handle the high-resolution timers and the normal jiffies-based timers. This can be seen by examining the code for the high-resolution timers in the `hrtimer_run_queues()` function defined in `kernel/hrtimer.c` or the code for the standard timers in function `__run_timers()` defined in `kernel/timer.c`.



- Two functions defined in `linux/delay.h`:
 - Microsecond delay: `udelay()`
 - Nanosecond delay: `ndelay()`
- These functions operate in a tight loop
 - Cannot be called with values greater than 20000
- Typical use:
 - Small delays needed by device drivers for slow devices

The following code fragment from `drivers/char/istallion.c` demonstrates how `udelay()` is used in device driver:

```
static void stli_ecpinit(stlibrd_t *brdp)
{
    unsigned long    memconf;

    outb(ECP_ATSTOP, (brdp->iobase + ECP_ATCONFR));
    udelay(10);
    outb(ECP_ATDISABLE, (brdp->iobase + ECP_ATCONFR));
    udelay(100);

    memconf = (brdp->memaddr & ECP_ATADDRMASK) >> ECP_ATADDRSHFT;
    outb(memconf, (brdp->iobase + ECP_ATMEMAR));
}
```



End of Lecture 13

- Questions and Answers
- Summary
 - Timing hardware and time sources
 - Real time access with `xtime`
 - Software timers (`struct timer_list`)
 - High-resolution timers (`struct hrtimer`)



Lab 13.1: Displaying Real Time

Deliverable: A device driver which displays the current time

Instructions:

1. Write a device driver that creates a `/dev/date` device file. When the file is read from it should display the current time with nanosecond resolution in the following format:

Current time in seconds: 1253853299.735441484

Don't start from scratch. Copy one of your lab solutions from the device driver lab to start working from.

Lab 13.2: Creating an Internal Interval Timer

Deliverable: A kernel module which displays a counter at a regular interval

Instructions:

1. Write a kernel module that initializes a counter to zero, then every five seconds increments the counter and print its value to the system log file using `printk()`.

Note: Be sure to cancel the interval timer gracefully when the module is unloaded.

Lab 13.1 Solutions

1. Solutions for the exercises in this lab can be found on the instructor machine at `/var/ftp/pub/rhd361-solutions/13-timing`.







Lecture 14

SystemTap

Upon completion of this unit, you should be able to:

- Configure SystemTap
- Use SystemTap to gain knowledge about kernel internals
- Know how to access the tapset library



- SystemTap simplifies the gathering of information about a running kernel
- Queried data is more accurate than interval-sampling tools
- Can help identify issues related to a performance or functional problem
- Greatly simplifies kernel instrumentation
 - No longer need kernel developer knowledge to collect kernel data!
- Requires `root` privileges since kernel modules are loaded
- Depends on the **kprobes** kernel subsystem

Historically, gathering information about the Linux kernel at runtime has been a highly complex task typically requiring deep knowledge about kernel internals. Tools such as OProfile and LTT (Linux Trace Toolkit) were created for that purpose and usually relied on sampling mechanisms in order to determine what the Linux kernel was doing; every 1 ms, for instance, a reading was taken and saved into a file for later examination. This type of analysis suffers from an inherent lack of precision for events occurring in between *interval* time periods.

SystemTap solves at least two issues associated with older instrumentation methods: precision and ease of use.

SystemTap was developed as a tool that would allow system administrators, who are not usually familiar with kernel internals, to gain a better knowledge about what the kernel executes on behalf of applications. Complexity is therefore greatly reduced.

SystemTap provides 100% reliable information about the running kernel because it does not rely upon any timing intervals. As a result, all kernel events, no matter how long they take to execute, can be monitored. This is a big improvement compared to a tool such as OProfile.

SystemTap was implemented on top of **kprobes**, a kernel subsystem that allows developers to attach code to any kernel function through the use of a kernel module. Using **kprobes** requires kernel development experience and skills. This is the reason why **kprobes** is entirely transparent from SystemTap's user interface.

Since modules need to be loaded, SystemTap requires `root`-privileged access to the system.

Three RPM packages are required to run SystemTap: `kernel-debuginfo`, `kernel-devel`, and `systemtap`.

WKS
System tap
St ap



- SystemTap scripting language
- The **stap** utility
 - Used to load scripts into the kernel
- Kernel modules compiled and loaded on-the-fly whenever a script is loaded
- The **tapset** script library
 - Facilitates script reuse and abstraction

The SystemTap scripting language provides a relatively simple interface to kernel instrumentation. Without this language, system administrators would have to code kernel modules manually, requiring much more advanced skills.

A SystemTap script can be as simple as a single line. For example, the following script places a probepoint on the kernel `sys_open()` function and prints all callers of the function with the function's arguments:

```
# stap -e 'probe syscall.open {printf("%s: %s\n", execname(), argstr)}'
```

The **stap** utility generally takes a few seconds to execute, as it achieves quite a few things in the background. It first parses the script passed on the command line (or as a file), and makes sure there are no syntax issues. It then generates a temporary binary kernel module from the script. The final step is to load the newly-generated module into the kernel. The module will remain loaded until the user terminates **stap** (generally by pressing *Ctrl-c* on the keyboard).

When the module is loaded, the message "starting probe" is printed to the console. Upon receiving *Ctrl-c*, the message "ending probe" is printed to the console, indicating a conclusion to the data collection.

After writing enough SystemTap analysis scripts for yourself, you may become known as an expert to your colleagues, who then may want to use your scripts. SystemTap makes it possible to share scripts in a controlled manner using libraries of scripts that build on each other. In fact, all of the functions (`pid()`, etc.) used in the scripts above come from tapset scripts already submitted by developers and system administrators. A "tapset" is just a script designed for reuse by installation into a special directory (`/usr/share/systemtap/tapset`) on the system.

The `execname()` function used in the short script above is a good example of a tapset function. It is already defined and may be used in your own scripts.



- Uses simple scripts similar to `awk`
 - Provides interface to the **kprobes** kernel subsystem
 - Probes use dotted notation and support wildcards
- Probe examples:
 - `probe kernel.function("foo")`
 - `probe kernel.function("*").return`
- The **stap** command compiles scripts into kernel modules
- Required packages
 - `systemtap`
 - `kernel-debuginfo` (must match *running* kernel)
 - `kernel-devel` (must match *running* kernel)

Historically, gathering information about the Linux kernel at runtime has been a highly complex task typically requiring deep knowledge about kernel internals. Tools such as OProfile and LTT (Linux Trace Toolkit) have been created for that purpose and usually relied on sampling mechanisms in order to determine what the Linux kernel was doing; every 1 ms, for instance, a reading was taken and saved into a file for later examination.

SystemTap was developed as a tool that would allow system administrators, who are not usually familiar with kernel internals, to gain a better knowledge about what the kernel executes on behalf of applications. Complexity is therefore greatly reduced.

As opposed to other mechanisms which rely on sampling at specific intervals of time, which suffers from an obvious lack of precision, SystemTap provides 100% reliable information about the running kernel. That means that all kernel events, no matter how long they take to execute, end up being monitored. This is a big improvement compared to a tool such as OProfile.

SystemTap was implemented on top of **kprobes**, a kernel subsystem that allows developers to attach code to any kernel function through the use of a kernel module. Using **kprobes** requires kernel development experience and skills. This is the reason why **kprobes** is entirely transparent from SystemTap's user interface.

Note: Outside of class, you can obtain debuginfo versions of the Red Hat Enterprise Linux kernel by enabling `/etc/yum.repos.d/rhel-debuginfo.repo`.



- Works in up to five passes
 1. Parse script
 2. Resolve symbols against *matching* kernel-debuginfo
 3. Translate into C
 4. Build kernel module
 5. Load module, run, and unload (requires root privileges!)
- Example: get a list of all kernel functions

```
stap -p2 -e 'probe kernel.function("*") {}' | sort -u
```

The SystemTap scripting language provides a relatively simple interface to kernel instrumentation. Without this language, system administrator would have to code kernel modules manually, which obviously requires much more advanced skills.

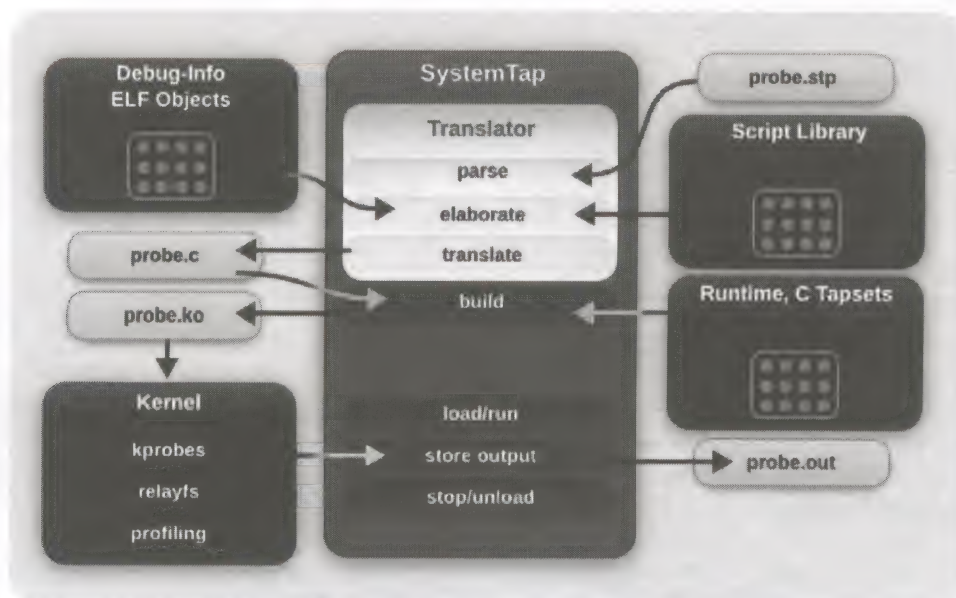
A SystemTap script can be as simple as a single line. For example, the following script places a probepoint on the kernel `sys_open()` function and prints all callers with the function's arguments:

```
$ stap -e 'probe syscall.open {printf("%s: %s\n", execname(), argstr)}'
```

The **stap** utility generally takes a few seconds to execute: it achieves quite a few things in the background. It first parses the script passed on the command line or as a file, and makes sure there are no syntax issues. It then generates a temporary binary kernel module out of the script. The next step is to load that module into the kernel. The module will then stay loaded, until the user terminates **stap** (generally by pressing CTRL-C on the keyboard).

After writing enough analysis scripts for yourself, you may become known as an expert to your colleagues, who will want to use your scripts. SystemTap makes it possible to share in a controlled manner; to build libraries of scripts that build on each other. In fact, all of the functions (`pid()`, etc.) used in the scripts above come from tapset scripts like that. A ``tapset" is just a script that is designed for reuse by installation into a special directory.

The **execname()** function used in the short script above is a good example of a tapset function. It is already defined and may be used in your own scripts.



SystemTap takes a compiler-oriented approach to generating instrumentation. The above diagram summarizes the flow of SystemTap implementation and data flow. In the upper right hand corner of the diagram is `probe.stp`, the example probe script written by the developer. This script is parsed by the SystemTap translator into parse trees and is checked for syntax errors. The translator then performs elaboration, pulling in additional code from the script library and determining locations of probe points and variables from the debug information. After the elaboration is complete the translator can generate a C-language loadable kernel module, `probe.c`.

The `probe.c` file is compiled into a regular kernel module, `probe.ko`, using the GCC compiler. The compilation may pull in support code from the runtime libraries. After GCC has generated `probe.ko`, the SystemTap daemon is started to collect the output of the instrumentation module. The instrumentation module is then loaded into the kernel, and data collection is started. Data from the instrumentation module is transferred to user-space via `relayfs` and displayed by the daemon. When the user enters `Ctrl-c` at the keyboard, the daemon unloads the module, which also shuts down the data collection process.

Red Hat hosts a wiki dedicated to SystemTap and the sharing of scripts at:

<http://sources.redhat.com/systemtap/wiki/>

The wiki also has a comparison of SystemTap versus DTrace:

<http://sources.redhat.com/systemtap/wiki/SystemtapDtraceComparison>

The architecture diagram shown above demonstrates five passes. In the first pass, the **stap** command is used to parse a hypothetical `probe.stp` script. The second pass uses the debug-info version of the kernel to resolve symbols. The third pass translates the script into c code. The fourth pass builds a kernel module from the c code. The fifth and final pass loads the kernel module, runs it, and saves its output to a hypothetical `probe.out` file.



- Tapset library scripts can be reused and shared
- Commonly used kernel probe points are exposed to all SystemTap scripts
- Tapset code library: `/usr/share/systemtap/tapset`
- Many probing points are already provided for:
 - I/O scheduler
 - Networking
 - NFS
 - Memory manager
 - Processes
 - SCSI
 - Signal subsystems
- Samples

`/usr/share/doc/systemtap-*/examples`

- **man -k systemtap**

Probing points are already provided for a variety of purposes. For example, performance monitoring and optimization is a common use for the following probe point examples:

<code>ioscheduler.elv_next_request</code>	Triggered when a request is retrieved from the request queue (for disks, a read or write).
<code>ioscheduler.elv_next_request.return</code>	Triggered when return from retrieving a request.
<code>netdev.receive</code>	Executed when data arrives on any network device.
<code>vm.pagefault</code>	Triggered upon a page fault (when memory is physically allocated, data retrieved from the swap device, etc).
<code>process.exec</code>	Triggered when a process attempts to invoke a new program.
<code>process.release</code>	Triggered when a process is released from the kernel (completely terminated, not in zombie state).
<code>tcp.sendmsg</code>	Triggered whenever the kernel sends a tcp message.

Sum of Sys report

not for .stp

was not for



- To trace entry of the `mkdir()` kernel function:

```
# stap -e 'probe kernel.function("sys_mkdir") { log ✓  
  ("enter") }'
```

- To trace exit of the `mkdir()` kernel function:

```
# stap -e 'probe kernel.function("sys_mkdir").return { log ✓  
  ("exit") }'
```

- To list the function names available to `stap`:

```
# stap -p2 -e 'probe kernel.function("*") {}' | sort | uniq
```



End of Lecture 14

- Questions and Answers
- Summary
 - In this unit, we explored using SystemTap to query information about the running kernel, for further analysis or identification of an underlying performance or functional issue.



Lab 14.1: Monitoring Context Switches with SystemTap

Scenario:

In this exercise, we will develop a SystemTap script which will let us monitor context switches that are invoked on the system.

Instructions:

1. Context switches take place whenever the kernel calls the `schedule()` function. Create a SystemTap script which will print a message whenever this function gets executed. Your script, named `csmon.stp`, should look as follows:

```
probe kernel.function("schedule").return {
    printf("Scheduler invoked")
}
```

Run it using the **stap** command:

```
[root@host ~]# stap csmon.stp
```

You may need to wait until the system starts another process. For instance, running the **top** command should produce some output.

Remember that a simple *Ctrl-c* will exit from SystemTap.

2. As you can see, the previous script generated several messages on the terminal. Obviously the `schedule()` function is commonly executed by the kernel.

Modify the `csmon.stp` script a bit so that it does not fill up the screen with messages:

```
global count

probe kernel.function("schedule").return {
    count++
    if ((count%1000)==0) {
        printf(".\n")
    }
    if (count==10000) {
        printf("reached 10000 context
switches!\n")
        count=0
    }
}
```

Run the script again with the **stap** command and see the difference. Much better!

3. It would be nice if we could get a report, say every 5 seconds, showing which processes were most actively executing context switches? That is precisely what the `timer` probe will let us do. Create a new script, named `cstop.stp` that look as follows:

```
global processes

function print_top () {
    cnt=0
    foreach ([name] in processes-) {
        printf("%-20s\t\t%5d\n",name,
processes[name])
        if (cnt++ == 20)
            break
    }
    printf("-----\n\n")
    delete processes
}

probe kernel.function("schedule").return {
    processes[execname()]++
}

probe timer.ms(5000) {
    print_top()
}
```

And then run it using the **stap** command. Which process generates the largest number of context switches on your system?

the scheduler

-
4. Modify the previous script so that it generates a list of 30 processes every 10 seconds. This modification should be a simple two line change. ✓
5. Modify your script so that it now generates a list of processes generating the largest number of calls to `sys_open`. This is particularly useful in order to find which processes constantly open new files, which can cause performance issues.

Your script should be almost identical to the last one and should only require a one line change.

fcntl,
q nme-panel







System and Kernel Initialization

Upon completion of this unit, you should be able to:

- Discuss the boot sequence
- Understand kernel initialization



- BIOS initialization
- Bootloader
- Kernel initialization
- **init** starts and enters desired run level by executing:
 - `/etc/rc.d/rc.sysinit`
 - `/etc/rc.d/rc` and `/etc/rc.d/rc[0-6].d/`
 - `/etc/rc.d/rc.local`
 - Virtual consoles
 - X Display Manager if appropriate

Each major step in a Linux system's boot sequence - BIOS initialization, bootloader, kernel initialization, and init startup - is covered in the upcoming pages.



- Power On Self Test (POST)
- Generate INT 19h (bootstrap)
- Peripherals detected
- Boot device selected
- First sector of boot device is loaded into memory
- "Magic number" is verified *AA55*
- Boot sector is executed

Here we discuss the BIOS and how it starts the boot process. The assumption is x86 platforms.

The BIOS (Basic Input/Output System) is the interface between the hardware and software on a very basic level. The BIOS provides the basic set of instructions used by the operating system. A successful boot depends on the BIOS, which in fact provides the lowest level of interface to peripheral devices and controls.

The BIOS will first run a power on self test (POST), then it will look for peripherals and a device to boot from. The hardware configuration information is permanently stored in a small area (usually 64 bytes) of CMOS (Complementary Metal Oxide Semiconductor), most commonly referred to as simply "the CMOS". The CMOS is powered by a small battery located in your motherboard. This battery allows the CMOS to retain it's settings even when the computer is turned off and disconnected from power.

At the end of the POST, a boot device is selected from the union of configured and detected boot devices. Any modern BIOS will allow you to set the desired order of preference for the boot devices from a list. Boot devices could include: the floppy drive, hard drive, USB drive, CDROM, network-interface, or other removable media).

From the first selected boot device, the Master Boot Record (or MBR, the first 512-byte sector of the device) is loaded into memory at offset 0x7C00. The MBR contains a small program (the primary bootloader) in the first 446 bytes, the partition table in the next 64 bytes, and the MBR signature in the last 2 bytes.

The BIOS verifies that the "magic number" 0xAA55 is in the MBR signature, and if so, the BIOS transfers control to the first memory location of the MBR bootloader code, which must be an instruction. If not, the BIOS may print a message and/or try the next boot device.



- The program loaded by the firmware bootstrap process
- The program that loads the "first program"
- Allows choice of boot image, specification of boot parameters, etc.
- Adds another layer to the boot process, but:
 - Allows multiboot configurations
 - Increases flexibility
- Default: GRUB "the GRand Unified Bootloader"

The boot manager is used to provide flexibility at boot time by providing a mechanism by which the system can be directed to use a different set of kernel parameters, or maybe even an entirely different kernel to use.



- Image selection
 - Select with space followed by up/down arrows on the boot splash screen
- Change an existing stanza in menu editing mode
- Issue boot commands interactively on the GRUB command line
- Kernel parameter passing
 - `Documentation/kernel-parameters.txt`
 - `parse_options()` parses and acts on some
 - Remaining are passed to the `init` process as arguments

The GRUB Boot Screen

When GRUB starts up, a graphical splash screen can be accessed by pressing *Return*; or *Space*. This screen has a list of menu entries, normally bootable images. You can select between the different images with the up and down arrow keys, and press *Return* to select a particular entry for booting.

If you want to pass arguments to boot images through menu editing mode or access the GRUB command line, and a GRUB password is set, you will need to type `p` followed by your GRUB password.

Menu Editing Mode

If you then select an entry and type `e`, you will be dropped into menu editing mode. This mode allows you to modify an existing boot stanza to pass options to the kernel or `init`, or select alternate root file systems or kernel files than you have configured in your existing stanzas. You can use arrow keys to select a line, `e` to edit a line, `d` to delete a line, `o` to add a line, and `b` to boot. For example, to boot into runlevel 2, you could select menu editing mode, select your Linux boot stanza, add a 2 to the end of your kernel line, and type `b` to boot the modified menu entry.

Kernel Parameters

Kernel parameters that can be specified are listed in `Documentation/kernel-parameters.txt`.

The GRUB Command Line

GRUB provides a command-line interface which can be used to write a temporary boot command from scratch, view the contents of files on the filesystem, perform diagnostic tests, or experiment with GRUB configurations. Most commands supported by the configuration file are available for interactive use. Editing commands are similar to those used by the bash shell, and *Tab* completion is available. If GRUB is not able to find a valid `grub.conf` file, it will default to the command line.

To exit menu editing mode or the command line and go back to the main GRUB menu, type *Esc*.

For more information about GRUB and `grub.conf`, look at **info grub**.



- Staged:
 - 1st Stage - small, resides in MBR or boot sector
 - "1.5 Stage" - provides understanding of filesystem types
 - `/boot/grub/fstype_stage1_5`
 - 2nd Stage - loaded from boot partition
- Minimum specifications for Linux:
 - Title, kernel location, OS root filesystem and location of the initial ramdisk (*initrd*)
- Minimum specification for other OS:
 - Title, boot device

The bootloader is responsible for loading and starting your Linux operating system (or possibly other operating systems) when the computer is started up.

The bootloader is generally invoked in one of two ways:

- BIOS passes control to an initial program loader (IPL) installed within a drive's Master Boot Record.
- BIOS passes control to another bootloader, which passes control to an IPL installed within a partition's boot sector.

In either case, the IPL (initial program loader) must exist within a very small space, no larger than 446 bytes. Therefore, the IPL for GRUB is merely a first stage, whose sole task is to locate and load a second stage bootloader, which does most of the work to boot the system.

There are two possible ways to configure bootloaders:

- primary bootloader: Install the first stage of your Linux bootloader into the Master Boot Record. The bootloader must be configured to pass control to any other desired operating systems.
- secondary bootloader: Install the first stage of your Linux bootloader into the boot sector of some partition. Another bootloader must be installed into the MBR, and configured to pass control to your Linux bootloader.



- To mount the root filesystem, the kernel typically needs to load modules
 - Examples: ext3, jbd, raid1, scsi_mod
- An *initial RAM disk* provides modules
 - Compressed cpio archive containing modules, other material
 - Created at install time
 - Specific to a particular hardware and software platform
 - Made available to the kernel by GRUB
- Use **mkinitrd** to rebuild
 - `mkinitrd /boot/initrd-$(uname -r).img $(uname -r)`

The modular nature of the Linux kernel presents a boot time issue:

The Linux kernel needs to mount the root filesystem. To do so, it typically needs access to modules. For example, the following filesystem-related capabilities are modular (not built into the basic kernel):

- ext3 filesystems and journaling for ext3 filesystems
- Logical volume management
- Software RAID
- SCSI controller support

If the kernel requires any of these capabilities to mount the root filesystem, it will need access to the related modules.

But this presents a quandary: where does the kernel get the modules? Typically, the kernel gets the modules from the `/lib/modules/$(uname -r)` directory. But, as the root filesystem is not yet mounted, it cannot access modules in this location. Hence, this presents a classic chicken/egg problem: the kernel cannot mount the root filesystem without access to the modules and the modules are on the root filesystem.

The GRUB bootloader and the kernel work together to provide the solution to this problem through the use of the initial RAM disk, which is part of the typical GRUB specification for a Linux kernel:

```
title Red Hat Enterprise Linux Server (2.6.18-53.el5)
    root (hd0,0)
    kernel /vmlinuz-2.6.18-53.el5 ro root=LABEL=/
    initrd /initrd-2.6.18-53.el5.img
```

The last line indicates that the initial RAM disk file is called `initrd-2.6.18-53.el5.img` (and that it is resident in `/boot`). Under Red Hat Enterprise Linux, this file contains a compressed cpio archive containing, among other things, modules needed to mount filesystems.

An initial RAM disk is specific to a particular hardware and software platform. Typically, it is created at install time and only those modules needed to mount filesystems for that system are included.

Initial RAM disks need to be recreated when filesystem hardware or software changes. For example, if a system has ext2 filesystems that have been converted to ext3 filesystems, the initial RAM disk will need to be recreated. To create an initial RAM disk, use the **mkinitrd** command:

```
mkinitrd /boot/initrd-$(uname -r).img $(uname -r)
```

The first argument is the path name to the initial RAM disk; the second argument is the directory within the `/lib/modules` directory that contains the modules. As initial RAM disks are specific to particular kernels, it is essential that the modules are appropriate for that kernel.

```
mkinitrd /boot/initrd-2.6.18-53.1.4.el5.img 2.6.18-53.1.4.el5
```

Manually specify the kernel version number if you are currently booted under a different version, as done above.

It is also possible to force a particular module to be placed in an initial RAM disk. This may be useful if, for example, you intend to make changes that you have not yet implemented. Two methods can be used to force a module into the initial RAM disk:

1. Using the `--with` option:

```
mkinitrd --with=scsi_mod /boot/initrd-$(uname -r).img $(uname -r)
```

Neither the `.ko` suffix nor the full path name to the module are necessary.

2. Placing a filesystem-related directive in `/etc/modprobe.conf`:

```
echo "alias scsi_hostadapter qla2300" >> /etc/modprobe.conf
mkinitrd /boot/initrd-$(uname -r).img $(uname -r)
```

The commands above will result in at least the `qla2300` module (plus any dependent modules) to be loaded into the initial RAM disk.



- GRUB
 - Command-line interface available at boot prompt
 - Boot from ext2/ext3, ReiserFS, JFS, FAT, minix, or FFS file systems
 - Supports MD5 password protection
- /boot/grub/grub.conf
- Changes to grub.conf take effect immediately
- If MBR on /dev/hda is corrupted, reinstall the first stage bootloader with:
 - /sbin/grub-install /dev/hda

/boot/grub/grub.conf has a format of global options followed by boot stanzas. Here is a sample grub.conf:

```
timeout=5
splashimage=(hd0,0)/grub/splash.xpm
hiddenmenu
password --md5 $1$/iX9y$Bk4yt37Ch2fZ5GFA
default=0

title Red Hat Enterprise Linux Server (2.6.18-53.el5)
    root (hd0,0)
    kernel /vmlinuz-2.6.18-53.el5 ro root=/dev/VolGroup00/LogVol00 rhgb
    quiet
    initrd /initrd-2.6.18-53.el5.img

title Windows XP Pro
    rootnoverify (hd0,1)
    chainloader +1
```

Changes to grub.conf take effect immediately. GRUB reads the configuration file at boot time, so the grub.conf file must be stored on a filesystem GRUB understands. These include ext2/ext3, reiserfs, FAT, minix, and FFS.

The MD5 password hash can be created with **grub-md5-crypt**.

If for some reason your MBR becomes corrupted and you need to reinstall GRUB, you can do so with the command **/sbin/grub-install boot-device**. Occasionally it may prove necessary for the user to set up grub manually. If **grub-install** fails for some reason try the following:

```
[root@stationX ~]# grub
... output omitted ...
grub> root (hd0,0)
grub> setup (hd0)
grub> quit
```




- Kernel boot time functions
 - Device detection
 - Device driver initialization
 - Mounts root filesystem read only
 - Loads initial process (init)

The kernel initialization files generate good output, but scroll by quickly. A good way to examine this output is to view `/var/log/dmesg`, which contains a snapshot of these kernel messages taken just after control is passed to `init`. Review of this output will reveal the basic initialization steps of the Linux kernel.

Device drivers compiled into the kernel are called, and will attempt to locate their corresponding devices. If successful in locating the device, the driver will initialize and usually log output to the kernel message buffer.

If essential (needed for boot) drivers have been compiled as modules instead of into the kernel, then they must be included in an `initrd` image, which is then temporarily mounted by the kernel on a RAM disk to make the modules available for the initialization process.

After all the essential drivers are loaded, the kernel will mount the root filesystem read-only.

The first process is then loaded (`init`) and control is passed from the kernel to that process.



- Used to specify resources that can be freed up after the kernel initializes
- Function/data is relocated to `.init.text` or `.init.data`
- `.init.text` and `.init.data` are freed by the kernel after boot completes
- `__init`
 - Used to mark functions as *initialization* functions
 - Used immediately before function name or between closing parenthesis and semicolon
 - Examples:

```
static void __init initfxn1(int a, int b)
{
    extern int c; c = a + b;
}

extern int initfx2(int, int) __init;
```

- `__initdata`
 - Used to mark initialized data as *initialization* data
 - Used between variable name and equal ("=") sign
 - Example:

```
static int initvar __initdata = 10;
```

Before stepping through the sequence of functions used during kernel initialization, an important concept that is used throughout the process is that of marking functions and data that will not be needed after initialization completes for removal. The `__init` and `__initdata` macros relocate their function/data into the `.init.text` and `.init.data`, which are later freed by the kernel (`free_initmem()`) after the boot process has completed.

Memory that was freed from `.init.text` and `.init.data` can be viewed in the output of **dmesg**:

```
# dmesg | egrep 'Freeing unused|Memory:'
Memory: 2071072k/2096576k available (2081k kernel code, 24240k reserved, 869k data, 220k init, 1179072k highmem)
Freeing unused kernel memory: 220k freed
```

See `include/linux/init.h` for more details.

Memory for kernel



- Macro mechanism for adding a function to kernel initialization

```
#define __define_initcall(level,fn) \
static initcall_t __initcall_##fn __attribute_used__ \
__attribute__((__section__(".initcall" level ".init"))) = fn
```

- Subsections provide processing order of initialization functions
- Seven levels of initialization + two application-specific sections

```
#define core_initcall(fn)          __define_initcall("1",fn)
#define postcore_initcall(fn)     __define_initcall("2",fn)
#define arch_initcall(fn)         __define_initcall("3",fn)
#define subsys_initcall(fn)       __define_initcall("4",fn)
#define fs_initcall(fn)           __define_initcall("5",fn)
#define device_initcall(fn)       __define_initcall("6",fn)
#define late_initcall(fn)         __define_initcall("7",fn)
```

```
#define console_initcall(fn) \
static initcall_t __initcall_##fn __attribute_used__ \
__attribute__((__section__(".con_initcall.init"))) = fn
```

```
#define security_initcall(fn) \
static initcall_t __initcall_##fn __attribute_used__ \
__attribute__((__section__(".security_initcall.init"))) = fn
```

- Ordering within subsections is determined by link order
- 2.4 kernel backwards compatibility

```
#define __initcall(fn) device_initcall(fn)
```

- `__initcall` uses a *reference*, whereas `__init` *relocates* the function to `.init.text`

The 2.4 kernel allowed a function to be marked with the `__initcall` macro, which would place a reference to the function in a special section of the kernel, `.initcall.init`, thereby adding the function to kernel initialization. At boot time, each function is processed in link order.

The 2.6 kernel improved upon the process by creating ordered subsections, providing some control over how the initialization functions are placed in the `initcall` section. Initialization functions can now be assigned priority within the initialization framework. The different subsections are defined in `include/linux/init.h`. At boot time, each subsection is processed in order (functions within a subsection are still processed by link order). The 2.4 `__initcall` macro still exists, but is now mapped to the `device_initcall()` subsection. There are 7 levels of initialization plus application-specific sections, each of which is designated an initialization priority.

Each named section is converted to a numbered text section. Each of the numbered text sections is then linked into the kernel in numeric order, as defined by a linker script (e.g. `arch/i386/kernel/vmlinux.lds.S`):

```
.initcall.init : AT(ADDR(.initcall.init) - LOAD_OFFSET) {
    __initcall_start = .;
```



```

    *(.initcall11.init)
    *(.initcall12.init)
    *(.initcall13.init)
    *(.initcall14.init)
    *(.initcall15.init)
    *(.initcall16.init)
    *(.initcall17.init)
    __initcall_end = .;
}

```

The `__init` and `__initdata` attributes *relocate* (not reference), to the `.init.text` and `.init.data` sections, which are freed by the kernel after boot time. The `__init` and `__initcall` macros can be used together, but care must be taken that functions marked with `__initcall` are not called by another method.



- `init/main.c`
 - `start_kernel()`
 - `rest_init()`
 - `init()`
 - `do_basic_setup()`
 - `do_initcalls()`
 - `init_post()`
- **init**

While the previous slide provides an overview of kernel initialization that is often good enough for the system administration level, this slide shows the initialization process at a deeper level, highlighting the most important function calls in the process. We will cover each, in turn, in even greater detail over the next few slides.



- When the kernel code loads, it first executes `start_kernel()`
- `__init` macro'd function
- Flow:

```
...
lock_kernel();
printk(linux_banner);
...
sched_init();
...
parse_early_param();
parse_args(...);
...
init_IRQ();
init_timers();
softirq_init();
...
profile_init();
...
console_init();
...
mem_init();
...
security_init();
...
signals_init();
...
rest_init();
```

`__init` functions have their memory released after initialization is completed (more specifically, when `init_post()` is executed).

`start_kernel()` spawns the `init` kernel thread.

`start_kernel()` eventually becomes process 0, the *idle process*, and never returns.



- Not an `__init` macro'd function
- Flow:

```
kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND);
numa_default_policy();
unlock_kernel();
preempt_enable_no_resched();
schedule();
preempt_disable();
cpu_idle();
```

The `init` kernel thread is initialized in `rest_init`, a non-`__init`'d function, to avoid the possibility that `free_initmem` (in `init_post`) might cleanse `start_kernel` of its memory segments before it has a chance to `cpu_idle`.

The Big Kernel Lock (BKL) is released.

`schedule` must be executed at least once to get the scheduling mechanism going.

Preemption is disabled before executing `cpu_idle`, forever rendering the *idle process* (process 0), idle.



- `__init` macro'd function
- Begins as a kernel thread
- Flow:

```
lock_kernel()  
...  
smp_init();  
...  
populate_rootfs();  
do_basic_setup();  
...  
init_post();
```

`init_post()` is used to remove `__init` and `__initdata` segments from memory.

The `init` kernel thread eventually becomes a user-level process, as determined by `init_post()`.



- `__init` macro'd function
- Flow:

```
init_workqueues();  
usermodhelper_init();  
driver_init();  
sysctl_init();  
do_initcalls();
```

At this point, the machine is initialized: the CPU subsystem (including SMP) is functional, and memory and process management is active.

`sysctl_init()` registers the `/proc` pseudo filesystem and presents to the kernel all of the tunable parameters specified in the `/proc/sys` structure.

Device drivers will then send hotplug events and be initialized here.



- Not an `__init` macro'd function(!)
- Flow:

```
free_initmem();
unlock_kernel();
...
system_state = SYSTEM_RUNNING;
...
sys_open(...) "/dev/console", O_RDWR, 0)
...
/*
 * We try each of these until one succeeds.
 *
 * The Bourne shell can be used instead of init if we are
 * trying to recover a really broken machine.
 */
if (execute_command) {
    run_init_process(execute_command);
    printk(KERN_WARNING "Failed to execute %s.  \n"
    Attempting "
                                "defaults...\n", \n"
    execute_command);
}
run_init_process("/sbin/init");
run_init_process("/etc/init");
run_init_process("/bin/init");
run_init_process("/bin/sh");

panic("No init found.  Try passing init= option to \n"
    kernel.");
```

Because this is the function where `__init` functions and `__initdata` is freed, this function must not be `__init`.

The system is marked to be running, and the console is opened for read-write.

By default, `init_post` then tries to execute the `init` process, but there is a fallback mechanism in place. First, if the `init=/path/to/program` kernel parameter has been set, the specified program becomes the `init` process (e.g. `init=/bin/sh`). If the kernel parameter has not been set, the default process to start is `/sbin/init`. If that executable errors (not found, or whatever), there is then a cascading mechanism that tries to execute other processes. Ultimately, if none of the processes listed can be executed, the kernel panics with an appropriate message.



- **init** reads its config: `/etc/inittab`
 - Initial run level
 - System initialization scripts
 - Run level specific script directories
 - Trap certain key sequences
 - Define UPS power fail / restore scripts
 - Spawn gettys on virtual consoles
 - Initialize X in run level 5

init

init is the parent of all processes. This is easily shown by running the **ps** command:

```
[student@stationX ~]$ ps -ef | grep init
init-+-apmd
|   -atd
|   -automount
|   -crond---crond
|   -deskguide_apple
|   -gdm-+-X
|       ^-gdm---gnome-session
```

Because **init** is the first process, it will always have a PID of number 1.

The file `/etc/inittab` contains the information on how **init** should set up the system in every run level, as well as the run level to use as default.

If the `/etc/inittab` file is missing or seriously corrupt, you will not be able to boot to any of the standard run levels (0-6) and will need to use single or emergency mode instead. This procedure is discussed in depth in the Troubleshooting unit of this course.



- `init` defines run levels 0-6, S, emergency
- The run level is selected by either
 - Default in `/etc/inittab` at boot

```
id:5:initdefault:
```

- Passing an argument from the bootloader
- Using the command `init new_runlevel`
- Show current and previous run levels
 - `/sbin/runlevel`

The following charts detail the run levels that Linux defines by default (there is a similar chart in the comments at the top of the `/etc/inittab` configuration file).

Bringing down your system:

0	halt - shutdown to a poweroff state - similar to shutdown -h now
6	reboot - shutdown and soft reboot - similar to shutdown -r now

Recovering your system:

1	single user mode - run <code>rc.sysinit</code>
s, S, single	alternate single user mode
emergency	bypass rc.sysinit , <code>sulogin</code> - prompts for root password

Normal operation of your system (choices for `initdefault`):

2	multiuser mode without NFS
3	full multiuser text mode, typical for servers
4	officially undefined
5	graphical login, typical for desktops and laptops

The first line in `/etc/inittab` defines the default runlevel, though it can be ignored by passing an argument to `init` directly after there is a prompt or via the kernel when passed as an argument in the bootloader.



End of Lecture 15

- Questions and Answers
- Summary
 - System BIOS
 - Boot loader - GRUB
 - Kernel initialization
 - `/sbin/init`



Lab 15.1: Exploring an Initial RAM Disk

Scenario: An initial RAM disk is an essential element in the boot up process. In this lab you will investigate the contents of an initial RAM disk, modify the contents, and then boot using the new RAM disk.

Deliverable: A system booting with a new RAM disk

Instructions:

1. Begin by identifying your current initial RAM disk file. This file will be in `/boot` and it will contain the release level of the kernel in the name. That is, the file will have the form: `/boot/initrd-version.img`. Display a long listing of this file on your system. *on RHEL-2.6.18-htj...*
2. Run the **file** command against this file to further identify it. Notice it is a **gzip** compressed file.

Warning: Do not modify the initial RAM disk file itself! Copy your initial RAM disk to your home directory so your original is not modified unintentionally.

Decompress the initial RAM disk file and run the **file** command against this data. You will see that the compressed file contains a **cpio** archive.

Hint: To accomplish this, consider the **zcat** command can read a stream from standard input and send the uncompressed data to standard output. Also consider the **file** command will read from standard input if given a dash (-) as an argument.

3. List the contents of the compressed **cpio** archive. Note that within the `lib` subdirectory there are a series of kernel objects. These should include `ext3.ko`, `jbd.ko`, and, depending on your hardware, may include other modules as well.

List the modules currently loaded into your running kernel. Modules are listed from most recently inserted to least recently inserted. Note that the modules on the bottom of the list are the modules that were loaded from your initial RAM disk.

4. Now create a new initial RAM disk file forcing the `raid1.ko` module to be included. Name the new initial RAM disk `/boot/initrd-raid1-version.img`. Note the `raid1.ko` module is not one of the modules in the original initial RAM disk file.

Warning: Be sure to give the new RAM disk a new name; do not overwrite your current initial RAM disk!

5. In the `/boot/grub/grub.conf` file, copy the stanza for your kernel (the `title`, `root`, `kernel`, and the `initrd` lines) so that you now have two versions of this entry. On the copy (without modifying the original stanza) modify the `initrd` line to use your new `initrd-raid1-version.img` file.

Reboot your computer, using the new initial RAM disk. Once again, list the modules in your currently running kernel. You should see the `raid1.ko` module now listed as one of the modules now available.

Mandatory cleanup: Reboot your computer such that you are using the original initial RAM disk.

Lab 15.1 Solutions

1. Begin by identifying your current initial RAM disk file. This file will be in `/boot` and it will contain the release level of the kernel in the name. That is, the file will have the form: `/boot/initrd-version.img`. Display a long listing of this file on your system:

```
[root@host ~]# KVER=$(uname -r) ; echo $KVER
2.6.18-164.el5
[root@host ~]# ls -l /boot/initrd-$KVER.img
```

2. Run the **file** command against this file to further identify it:

```
[root@host ~]# file /boot/initrd-$KVER.img
```

Warning: Do not modify the initial RAM disk file itself! Copy your initial RAM disk to your home directory so your original is not modified unintentionally:

```
[root@host ~]# cp /boot/initrd-$KVER.img ~
[root@host ~]# file initrd-$KVER.img
```

Decompress the initial RAM disk file and run the **file** command against this data:

```
[root@host ~]# zcat initrd-$KVER.img | file -
```

You will see that the compressed file contains a **cpio** archive.

3. List the contents of the compressed **cpio** archive. Note that within the `lib` subdirectory there are a series of kernel objects. These should include `ext3.ko`, `jbd.ko`, and, depending on your hardware, may include other modules as well.

```
[root@host ~]# zcat initrd-$KVER.img | cpio -itv ✓
| less
```

List the modules currently loaded into your running kernel. Note that the modules on the bottom of the list are the modules that were loaded from your initial RAM disk:

```
[root@host ~]# lsmod | less
```

4. Now, create a new initial RAM disk file, forcing the `raid1.ko` module to be included. Name the new initial RAM disk `/boot/initrd-raid1-version.img`. Note the `raid1.ko` module is not one of the modules in the original initial RAM disk file.

Warning: Be sure to give the new RAM disk a new name; do not overwrite your current initial RAM disk:

```
[root@host ~]# mkinitrd --with=raid1 ✓
/boot/initrd-raid1-$KVER.img $KVER
```

5. In the `/boot/grub/grub.conf` file, copy the stanza for your kernel (the `title`, `root`, `kernel`, and the `initrd` lines) so that you now have two versions of this entry. Modify the copy `initrd` line to use your new `initrd-raid1-version.img` file.

Your new stanza should look something like this:

```
title Red Hat Enterprise Linux Server w/ RAID1 ↵  
(2.6.18-164.el5)  
    root (hd0,0)  
    kernel /vmlinuz-2.6.18-164.el5 ro ↵  
root=/dev/vol0/root rhgb quiet  
initrd /initrd-raid1-2.6.18-164.el5.img
```

Reboot your computer, using the new initial RAM disk. Once again, list the modules in your currently running kernel. You should see the `raid1.ko` module now listed as one of the modules now available.

```
[root@host ~]# lsmod | less
```

Mandatory cleanup: Reboot your computer such that you are using the original initial RAM disk.





Kernel Debugging 2: Crash Dumps

Upon completion of this unit, you should be able to:

- Install and configure the `kdump/kexec` tool
- Describe kernel memory dump strategies available under Red Hat Enterprise Linux



- **Netdump** sends memory content to a remote machine
 - Network driver has to be supported and still active after a crash
 - Remote system has to be available at crash time
 - RHEL3 and RHEL4 only
- **Diskdump** copies memory to a local disk
 - Requires a supported (non-IDE) storage device
 - Driver has to be up and running after crash
- **kdump** replaces both **netdump** and **diskdump**
 - Introduced with the release of Red Hat Enterprise Linux 5
 - Supports network or disk-based memory captures

For many years, UNIX derivatives have written an image of kernel memory onto their swap media when the kernel panics. As part of the reboot procedure, these systems would first check the swap areas for a valid dump image and then try to copy the dump information from the swap area into another place in the regular file system. Then, when the system had finished rebooting, a programmer would be able to perform a post-mortem analysis using the saved kernel dump image.

Linux, on the other hand, was designed to run without any swap area. Lacking a dedicated swap area, Linux had no place to copy the kernel image without needing lots of filesystem and I/O support still working in the kernel.

Another reason that Linux lacked crash dump support was that it was generally expected to run on a small, inexpensive system with a user/programmer in close attendance. Embedded systems that used Linux were expected to provide watchdog timers to reboot the system should it hang or stop.

As Linux came to be used as an enterprise-level server, perhaps in clusters of dozens (if not hundreds) of machines, requiring that a human be in close contact with the system became impractical. This underscored Linux's problems:

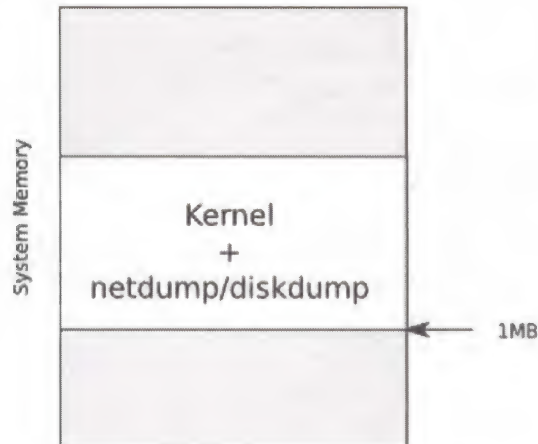
- Kernel status messages, such as `printk()` output, were permanently lost unless `syslog` and `klogd` had been able to drain the message buffer before the kernel crashed.
- Most Linux systems did not have a hard-copy console log, so the OOPS messages were frequently lost.
- Depending on the platform, a kernel panic might reboot the machine, or simply place the CPU into a tight loop.

Several tools have been added to remedy these problems. Under RHEL3 and RHEL4, the `netconsole` feature ensures that the console messages produced by a kernel panic are reliably sent to the `syslog` facility on a networked machine: this ensures that at least the OOPS report is available for analysis. The `netdump` feature uses an Ethernet device to send a copy of the kernel memory space to a networked machine, where it may be analyzed later. RHEL4 also supports `diskdump`, which makes it possible to dump memory to a specific storage device.

RHEL5 brings several improvements and more flexibility with `kexec` and `kdump`, which act as a replacement for `netdump` and `diskdump`.



- The problem with `netdump` and `diskdump`:
 - The tools exist within the faulted kernel we are trying to diagnose
 - The error could exist in the `netdump/diskdump` code, itself



The slide graphic represents the memory layout of the kernel in a traditional `netdump/diskdump` system. The kernel is loaded into system memory approximately 1MB into it. Inside the running kernel is the code that implements `netdump` and/or `diskdump`. This is problematic because it means that `netdump/diskdump` are part of the very program that took the error, and that an image is being taken of. As a result, there is a good chance that the error occurs in the very code that `netdump/diskdump` are going to use for the core image dump. When the tools used to diagnose the problem become part of the problem, analysis can become very difficult if not impossible.

www.redhat.com/rhel/details/limits



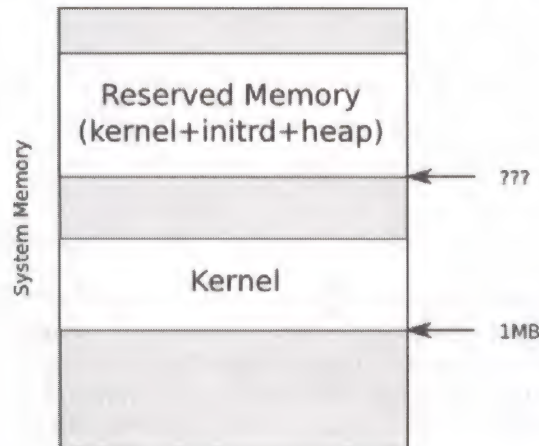
- Available in RHEL 5.0 onward
- Kernel core dump capturing facility
- Obsoletes `netdump`/`diskdump`
- Based on **kexec** infrastructure
- Supported on x86, x86_64, ia64, and ppc64 platforms
- As of RHEL5.1, the virtualized kernel (all but ia64) can use kdump

Kdump is a facility that relies on other newer technologies, such as kexec, to capture reliable core memory images from a crashed Linux kernel. Kdump obsoletes `netdump` and `diskdump` utilities.

Kdump works on x86, x86_64, ia64, and ppc64 platforms (though ppc64 requires a separate capture kernel, provided by the `kernel-kdump` package). As of RHEL 5.1, the virtualized kernel (x86, x86_64, but not ia64) can also use kdump.



- Use a secondary kdump kernel
 - Crash dump is captured from a freshly booted kernel
- Avoid going through BIOS, so crashed kernel's memory is preserved



Kdump improves on the `netdump/diskdump` model by using a new technology called `kexec`. `Kexec` allows us to reserve an area of kernel memory very early in the initial kernel boot process. In this reserved area of memory, we are able to load a kdump kernel. Because memory is reserved very early in the boot process, we can guarantee contiguous memory for our kdump kernel. Also loaded into the reserved memory area is a new `initrd.img` and some additional heap for our secondary kdump kernel.

This is an improvement because we can modify virtual memory page table access controls to guarantee no other process can write to our original kernel's memory. This has the advantage of guaranteeing the kernel we want to dump has not been touched in any way, resulting in a pristine kernel dump.

The three main technologies that make kdump possible are `kexec`, the ability to relocate the starting address of a kdump kernel, and in-place kernel decompression.



- Reserves system memory early in the boot process
 - kdump kernel
 - Special initrd image
 - Additional heap memory
- Fastboot mechanism
 - Boot fresh kernel from context of an already-running kernel
 - Reboot path through BIOS is unnecessary
 - Saves time for developers who are constantly booting a machine

Kexec eliminates the time-consuming and memory-clearing path through BIOS to load the new kernel.

Kdump improves on the `netdump/diskdump` model by using a new technology called kexec. Kexec allows us to reserve an area of kernel memory very early in the initial kernel boot process. In this reserved area of memory, we are able to load a kdump kernel. Because memory is reserved very early in the boot process, we can guarantee contiguous memory for our kdump kernel. Also loaded into the reserved memory area is a new `initrd.img` and some additional heap for our secondary kdump kernel.

This is an improvement because we can modify virtual memory page table access controls to guarantee no other process can write to our original kernel's memory. This has the advantage of guaranteeing the kernel we want to dump has not been touched in any way, resulting in a pristine kernel dump.

While purely an academic example (it isn't very practical to try this on a production machine), the following demonstrates how kexec can be used to switch to another kernel, without having to reinitialize or go through the BIOS. First, make sure kexec is installed:

```
# yum install kexec-tools
```

Now configure a reboot to "pivot" to the new, specified kernel:

```
# kexec -l /boot/vmlinuz-$(uname -r) --initrd=/boot/initrd-$(uname -r).img \
--command-line="`cat /proc/cmdline`"
```

At this point, rebooting the machine will cause it to switch to the new kernel and bypass the BIOS.

```
# reboot
```




- Traditionally an immovable kernel starting location
 - Approximately 1MB offset in system memory
 - Alternate kernels would load on top of the image we wished to capture
- Improved in older kdump versions
 - Define alternate starting point in memory for kdump kernel
 - Necessitated building and managing a different second kernel

Traditionally the bootloader loads the compressed kernel into memory, decompresses it into separate buffers, and then reassembles the uncompressed portions into a particular starting address. The starting memory location of the uncompressed kernel used to be hard coded at an offset of approximately 1MB. This prevented a kdump kernel from being loaded, as it could load only on top of the memory image we wished to capture.

The original kdump improved on this by allowing a different kernel image starting point in memory to be defined, e.g. 16MB, but this necessitated building and using a kdump kernel that is different from the running kernel that we are trying to analyze. While feasible, managing two different kernels makes system administration more difficult.



- Allows for single kernel image
 - Same image can be moved to any safe location
- Possible because of dynamic relocation
- Kernel now includes relocation information
- Dynamically updates location information during decompression

Another piece of enabling technology for kdump is the ability to do in-place kernel decompression. The bootloader loads the compressed kernel, the compressed kernel relocates itself to some area of memory deemed safe by an algorithm within the compressed kernel, and finally is decompressed inline. The new decompression method allows for a single kernel image (the image can be moved to any safe location).

In addition to decompression code that allows decompression in any location deemed safe, the kernel now maintains a database of relocation information at the end of the kernel image which, while decompressing, can dynamically update any accesses to relatively located data during decompression. This is how the kernel can exist in any location of memory that it chooses to. This technology has been around for a while (for example, in shared libraries and some user space applications), but its never been applied to the kernel before until now.

As a result, we can now load the *same* kernel in memory that we booted from, instead of having to load a second kernel.



- Sequence of events:
 1. System begins halting path that normally occurs at a panic call
 2. Instead of printing out a backtrace to the screen, it checks to see if the kdump kernel is loaded
 3. If no:
 1. Halt the system
 2. Print the normal backtrace to screen
 4. If yes:
 1. Unlock memory
 2. Halt the other CPUs (if multi-CPU system)
 3. Do a minimal reset of video and other devices as necessary
 4. Jump to the new kernel
 5. Start booting again
- Net effect is to boot a normal kernel but don't trap back into the BIOS

Upon kernel crash, the system invokes a trap, and proceeds down the halting path normally started at the time of a panic call. Instead of simply printing out a backtrace to the screen, it checks to see if the kdump kernel is loaded (at the same point traditionally used to check if `netdump/diskdump` are enabled). If there isn't a kdump image loaded, the system is halted as normal and a backtrace is printed to the screen.

If a kdump kernel is loaded, then memory is unlocked, all other cpus are halted (important on a multi-cpu system where other CPUs might still be actively trying to modify memory). A minimal reset of video and other devices that are necessary is performed, followed by a jump to the new kernel and beginning the boot process.

Bootting a kdump kernel has the net effect of booting a normal kernel with the important exception that it doesn't trap back into BIOS, which means memory isn't cleared. As a result, we have the ability to capture unaltered memory later on.



- Much more complex than normal initrd image
- Kdump's initrd image has a different goal
 - Detect hardware on the system
 - Load modules for copying core memory image to specified targets
 - Capture a memory image
 - Restart the machine back to normal mode of operation ASAP
- Doesn't perform "normal" initrd functions (e.g. starting services)
- Allows interactivity
 - Based on `busybox` instead of `nash`
 - Contains utilities for capturing kdump kernel

All the new functionality for kdump is all contained in its special initrd image. Normally `initrd.img` is responsible for finding and mounting the root filesystem and running all the init processes which start all the system services. In a kdump kernel, however, the goal is different: we don't want to start all of those services. What we want to do is capture a memory image and restart the machine back to its normal mode of operation as soon as possible.

To this end, a unique and complex initrd filesystem was built that has utilities specifically for the purpose of capturing the memory image and restarting the machine. The differences in a kdump boot procedure are therefore apparent once the `initrd.img` starts to load.

The kdump `initrd.img` will detect hardware on the system and load modules to copy the core memory image to the target specified in the kdump configuration file. Kernel core memory image dump targets can be modified to just about anything. Currently, an ssh server, NFS mount, local disk, etc. are supported.

The kdump `initrd.img` is based on `busybox` (instead of `nash`), so there is a wealth of tools available for interactively dumping a kernel image from a shell, for example.



1. Install the `kexec-tools` RPM package
 - Installation-time:
 - `Base...Optional packages`
 - Post-installation:
 - `# yum install kexec-tools`
2. Configure `/etc/kdump.conf` to specify the location where the vmcore should be dumped (see following slides)
 - Local system or another server via NFS or sshd
 - Only one dump target can be configured
3. Modify the kernel boot line in `/etc/grub` to reserve memory for the kdump kernel, then reboot using that kernel
 - Value format: `size@offset` (e.g. `crashkernel=128M@16M`)
 - 128MB minimum recommended
4. Enable kdump service and configure to persist a reboot
 - `# service kdump start; chkconfig kdump on`
5. Test using SysRq
 - `# echo c > /proc/sysrq-trigger`

Configure the file `/etc/kdump.conf` to specify the location where the kernel core file should be dumped. Kdump can dump to the local system (raw device or formatted filesystem) or server via SSH or NFS.

Upon system reboot, the amount of memory specified with the `crashkernel` kernel parameter is reserved for the kdump kernel, at the specified offset. The output of the command `free -m` should reflect this usage. Note: While it may be possible to use less than 128M, it may be unreliable and should be tested first. It is recommended that IA64 and PPC64 machines use a minimum of 256MB, though some larger enterprise server machines may require as much as 512MB. Again, test your machine for its requirements before putting it into production.

After a crash and reboot into the kdump kernel, the kernel core dump should be sent to the location specified in `/etc/kdump.conf` once the kdump service starts.

Note: when booting the kdump kernel, console frame-buffers and `x` are not properly supported. While the kernel core dump will proceed normally, the console video will likely be garbled if a kernel parameter such as `"vga=792"` is specified or `x` is started.



- Specify with appropriate line entry in `/etc/kdump.conf`
- Dumping to a raw (unformatted) partition:
 - `raw /dev/sdb1`
 - `vmcore` file is copied directly onto raw partition
 - Raw partition must be at least as large as system memory
- Dumping to a local filesystem (two different methods):
 - `path /local/pathname`
 - Default location: `/var/crash/date/`
 - `fs_type partition`
 - Partition may be device node, label, or UUID
 - `mount -t fs_type partition /mnt`
 - Copies `/proc/vmcore` to `/mnt/var/crash/date`

Any changes to the `/etc/kdump.conf` file are implemented by restarting the `kdump` service.

There are several examples of these parameters in `/etc/kdump.conf`.



- The client system must have write access to a NFS mount point
- Edit `/etc/kdump.conf`
 - `net host.example.com:/writable/export/directory`
- This will dump a core file to the directory:
 - `/writable/export/directory/var/crash/hostname-date/`

Kdump can be configured to copy the `vmcore` image to a remote location using NFS.



- Kdump can use `scp` to connect to a remote SSH server to dump its core file
- Create a user on the remote server
 - Configure write permissions for user on server's `/var/crash/`
- Edit `/etc/kdump.conf`
 - `net kdumpuser@crash.example.com`
- This will dump a core file to the directory:
 - `/var/crash/hostname-date/`
- Propagate SSH host keys to the remote server
 - `# service kdump propagate`

Kdump can be configured to copy the `vmcore` image to a remote location using `scp`.

The user configured for `ssh/scp` dumps should be able to connect to the remote service without a password and should have write permissions to the `/var/crash/` directory on the remote machine.

look at
propagate
for
ssh
key
exchange



- Enterprise-class machines can have large amounts of RAM
- Dumping the entire contents of that RAM (the default method) produces large crash dump files
 - Excessive storage space often required
 - Very large core files are unwieldy

Enterprise-class machines can have very large amounts of system memory installed. The crash dump files generated by these machines can be terabytes in size if the entire contents of the RAM is dumped to a file, resulting in unwieldy dump files and excessive amounts of storage space just to hold them.



- Dump file size can be reduced by specifying an alternate dump capture method
 - `core_collector` parameter in `/etc/kdump.conf`
- Most common alternate method chosen is `makedumpfile`
 - Can reduce the size of the file dumped through some combination of:
 - Dump filtering
 - Page-level compression
 - Provided by `kexec-tools` package
 - Requires `kernel-debuginfo` package corresponding to running kernel
 - Needs `vmlinux` for structure and symbol information
 - Cannot be used with `ssh/scp` or NFS dumps
 - Example:
 - `core_collector makedumpfile -c`

An alternate method of for capturing a kernel dump file can be specified in the `/etc/kdump.conf` configuration file. The most common reason for doing so is to specify the `makedumpfile` method, which can dramatically reduce the size of the resultant crash dump file by filtering unnecessary pages from it, and compressing the rest.

The `makedumpfile` method is provided by the `kexec-tools` package, and requires that the `kernel-debuginfo` package corresponding to the currently running version of the kernel also be installed. The `kernel-debuginfo` package is needed because `makedumpfile` requires a matching `vmlinux`, with its compiled with debug information, to acquire information such as data structure sizes, member offsets, symbol addresses, etc.

The `makedumpfile` method cannot be used with `ssh/scp` or NFS remote dump location specifications, nor with the `path` local filesystem specifier. However, a local location *must* be specified, so the alternate form of local filesystem specification (see previous slide) must be used in `/etc/kdump.conf`. For example (see the comments in `/etc/kdump.conf` for more details):

```
ext3 LABEL=/kdumps
```

An example capture method specified in `/etc/kdump.conf` might be:

```
core_collector makedumpfile -c
```

where the `-c` option enables the compression function of each page.

For more options associated with the `makedumpfile` method, see the output of `makedumpfile --help`.



- **makedumpfile** can be used to filter out some combination of unnecessary pages:
 - Pages filled with zero
 - Cache pages
 - User process data pages
 - Free pages
- `-d dump_level` option

Dump filtering presumes that not all of the elements of RAM are needed for a successful core dump analysis.

makedumpfile is a user-space command that can be used to weed out pages from the kernel dump file that are not necessary for post-mortem debugging and analysis by examining memory management data structures in the core dump to identify certain page types that are eligible for removal.

Examples of pages that can be filtered out of the kernel dump file:

Pages filled with zero. The **makedumpfile** command actually searches the entire kernel dump file looking for pages that contain only zeroes. The page can be removed, but the crash dump analysis tools still return a zero when the page is accessed.

Cache pages. The **makedumpfile** reads all pages in the crash dump. Cache pages are identified and filtered out by first determining the memory model, then the `struct page`, and then the attributes of entries within the `struct page`. Specifically, if it is determined that the `PG_swapcache` bit of `struct page` attribute `flags` is on (and therefore is part of the swap cache), or if the `PG_lru` bit of `struct page` attribute `flags` is set to on and the `PAGE_MAPPING_ANON` bit of `struct page` attribute `mapping` is off (and therefore is part of the page cache), then the page is a cache page, and is eligible for being removed from the crash dump.

User process data pages. The **makedumpfile** command reads all pages in the crash dump, and if any pages have the `PAGE_MAPPING_ANON` bit of `struct page` attribute `mapping` turned on, then they are user process data pages and are eligible for removal from the crash dump.

Free pages. The **makedumpfile** command reads all pages in the crash dump and looks to see if any page's `free_area` attribute of `struct zone` is linked, and if so, becomes eligible for removal.

Which pages are filtered out depends up the dump level specified with the `-d dump_level` option to **makedumpfile**. See the output of **makedumpfile --help** for more information.

The effectiveness of the filtering depends largely upon what the production kernel was doing at the time of the crash. If the machine was not up for very long when the crash occurred, there is a good chance much of the memory is free pages, and so the resultant file size will be much smaller than the unfiltered version.



- Ordinary compression tools can reduce crash dump file size
 - ...but it will need to be decompressed for analysis
- **makedumpfile** does page-level compression:
 - Pages are decompressed on the fly when accessed
 - Minimizes file size, even during analysis
 - `kdump`-compressed format, by default
 - Directly readable by **crash**
- `-c` option

Core dump files can be reduced in size through ordinary compression tools (e.g. **gzip**), but then they need to be decompressed somewhere in order to be analyzed.

The advantage of page-level compression is that individual pages can be decompressed when an analysis tool accesses it, thereby reducing the size of the file even during analysis.

By default, the **makedumpfile** command creates filtered/compressed crash dump files in the `kdump`-compressed format, which can be analyzed by **crash** without alteration.

makedumpfile can also create crash dump files in the ELF format (readable by both **crash** and **gdb**), but the ELF format does not support compressed dump files.

Page-level compression can be specified with the `-c` option to **makedumpfile** in `/etc/kdump.conf`. See the output of **makedumpfile --help** for more information.



- Kdump is highly architecture specific
 - Unlike `netdump/diskdump`
 - Testing is a must for good coverage
 - x86 systems have different needs than x86_64 systems, for example
- Drivers aren't used to being reset without BIOS intervention
 - Drivers often assume BIOS preps hardware for use
 - BIOS reset doesn't occur when booting kdump kernel
 - Drivers must be properly written to reset to a known state from a previously unknown state
 - Don't assume it just works...test it!
- Memory intensive
 - Must reserve enough memory for kernel, initrd, and additional heap space
 - Too little and risk out-of-memory condition when it boots

Kdump is highly architecture-specific (unlike `netdump/diskdump`), and so needs lots of testing for good coverage. For example, x86 systems have different needs than x86_64 systems that have to be taken into account.

"Pivoting" to the kdump kernel can cause problems with drivers that aren't used to being reset without BIOS intervention. The BIOS normally resets a lot of the hardware at boot time, and drivers often assume that that's done. When booting from a kdump kernel, that BIOS reset doesn't happen, and so drivers might need to be altered to make sure they properly reset their hardware and come back up into a known state from an unknown state.

An amount of memory (e.g. 128MB of RAM) must be reserved by the first kernel at boot time so that there is a safe area for the second, kdump kernel to boot. The amount of memory reserved must be adequate hold the second kernel, its initrd image, and the additional heap space. Less than this and you could run into out-of-memory conditions during which processes could be killed and the system could halt. To compute the actual minimum memory requirements for a system, refer to <http://www.redhat.com/rhel/compare/> for the listed minimum memory requirements and add the amount of memory used by kdump to determine the actual minimum memory requirements.



End of Lecture 16

- Questions and Answers
- Summary
 - Using `kdump/kexec` to generate kernel core dumps

Lab 16.1: Implementing Kdump/Kexec

Scenario: In this exercise, you will implement kdump/kexec and explore some options in `/etc/kdump.conf`.

Deliverable: One or more crash files for analysis

Instructions:

1. Install the `kernel-debuginfo` and `kexec-tools` packages if necessary.
2. Do not edit `/etc/kdump.conf` at this time. Modify the kernel boot line in `/etc/grub.conf` to reserve 128MB of memory at boot time with an offset of 16MB, then reboot into that kernel.
3. Once the machine has finished rebooting, start the kdump service and make sure it persists a reboot of the machine.
4. Initiate a crash of the system to test the configuration and examine the sequence of events.
5. Examine the `/var/crash` directory for the name and size of the core dump.
6. What happens if you do not specify a large enough area of memory at boot time for the kernel? Replace `crashkernel=128M@16M` with `crashkernel=16M@16M` and force a system crash.

After testing, be sure to restore the proper memory reservation and reboot.

7. The core dump file is fairly large. Modify `/etc/kdump.conf` with an appropriate local filesystem specification and an alternate dump capture method to use page compression. Restart the kdump service to put the changes into effect.
 8. Initiate a system crash to test the configuration.
 9. After the system dumps core and reboots back into the default kernel, examine the contents of `/kdump/var/crash`. How does the size of the core dump file compare with the size of the previous core dump file?
-
10. The core dump file probably still contains elements that aren't needed for our analysis. Further reduce the size of the core dump by filtering out as many unneeded elements as possible and compare the resultant core dump size to the original.
 11. Experiment and use **crash** to analyze the core dumps you created.

Lab 16.1 Solutions

1. Install the `kernel-debuginfo` and `kexec-tools` packages if necessary.

Verify installation:

```
[root@host ~]# rpm -q kernel-debuginfo ✓  
kexec-tools
```

If not:

```
[root@host ~]# yum -y install kernel-debuginfo ✓  
kexec-tools
```

2. Do not edit `/etc/kdump.conf` at this time. Modify the kernel boot line in `/etc/grub.conf` to reserve 128MB of memory at boot time with an offset of 16MB, then reboot into that kernel.

```
[root@host ~]# vi /etc/grub.conf
```

Append the parameter `crashkernel=128M@16M` to the kernel line that will be used by default, then reboot.

3. Once the machine has finished rebooting, start the `kdump` service and make sure it persists a reboot of the machine:

```
[root@host ~]# service kdump start; chkconfig ✓  
kdump on
```

4. Initiate a crash of the system to test the configuration and examine the sequence of events:

```
[root@host ~]# echo c > /proc/sysrq-trigger
```

Shortly after the forced crash of the system, you should observe the system booting into an alternate kernel (without first having gone through the BIOS). Don't worry if X looks broken -- it isn't supported in the `kdump` kernel. After a while, the system will automatically reboot again, this time going through the BIOS and booting into the default kernel as specified in `grub.conf`.

5. Examine the `/var/crash` directory for the name and size of the core dump.
6. Replace `crashkernel=128M@16M` with `crashkernel=16M@16M` and force a system crash. If you do not specify a large enough area of memory at boot time for the kernel, the system will hang at the crash and proceed no further.

After testing, be sure to restore the proper memory reservation and reboot.

7. The core dump file is fairly large. Modify `/etc/kdump.conf` with an appropriate local filesystem specification and an alternate dump capture method to use page compression. Restart the `kdump` service to put the changes into effect.

Note: In order to use page compression the `makedumpfile` dump capture method must be specified. When doing so, it *requires* a local filesystem be specified. The `path` method cannot be used here, so we must create a new filesystem and refer to its device in `/etc/kdump.conf`.

Create a new 10GB partition, `ext3`-formatted, with a label of `/kdumps` (Note: the following example assumes your primary drive is named `/dev/sda`):

```
[root@host ~]# fdisk /dev/sda
```

Create an extended partition that contains the remainder of the disk, then create the 10GB partition (`/dev/sda5`).

```
[root@host ~]# partprobe /dev/sda
[root@host ~]# mke2fs -j -L /kdumps /dev/sda5
[root@host ~]# mkdir /kdumps
[root@host ~]# echo "LABEL=/kdumps /kdumps ext3
defaults 0 2" >> /etc/fstab
```

Edit `/etc/kdump.conf`:

```
[root@host ~]# vi /etc/kdump.conf
```

Add the following lines to the bottom of the file:

```
core_collector makedumpfile -c
ext3 LABEL=/kdumps
```

Restart the `kdump` service and make sure it completes successfully.

```
[root@host ~]# service kdump restart
```

8. Initiate a system crash to test the configuration:

```
[root@host ~]# echo c > /proc/sysrq-trigger
```

9. After the system dumps core and reboots back into the default kernel, examine the contents of `/kdumps/var/crash`. The core dump should be a smaller file than before.
10. The core dump file probably still contains elements that aren't needed for our analysis. Further reduce the size of the core dump by filtering out as many unneeded elements as possible. Edit `/etc/kdump.conf` and modify the `makedumpfile` line near the bottom of the file to read:

```
core_collector makedumpfile -c -d 31
ext3 LABEL=/kdumps
```

Restart the `kdump` service and make sure it completes successfully:

```
[root@host ~]# service kdump restart
```

Force a system crash and verify the results. The final core dump file size should be even smaller than with page-level compression.





Lecture 17

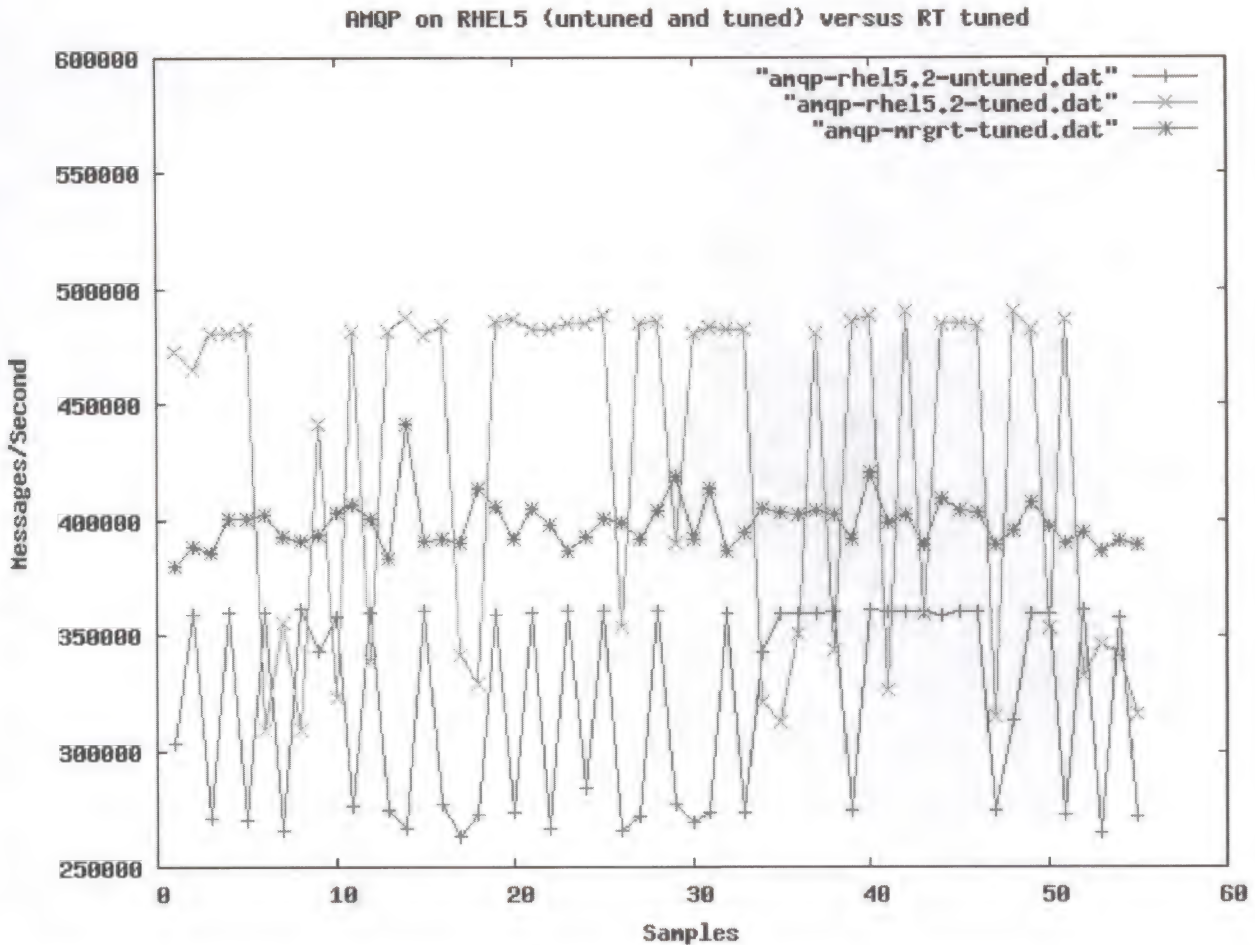
Red Hat Enterprise Linux Realtime Kernel

Upon completion of this unit, you should be able to:

- Understand goals of the realtime kernel
- Use tuning tools provided with the realtime kernel

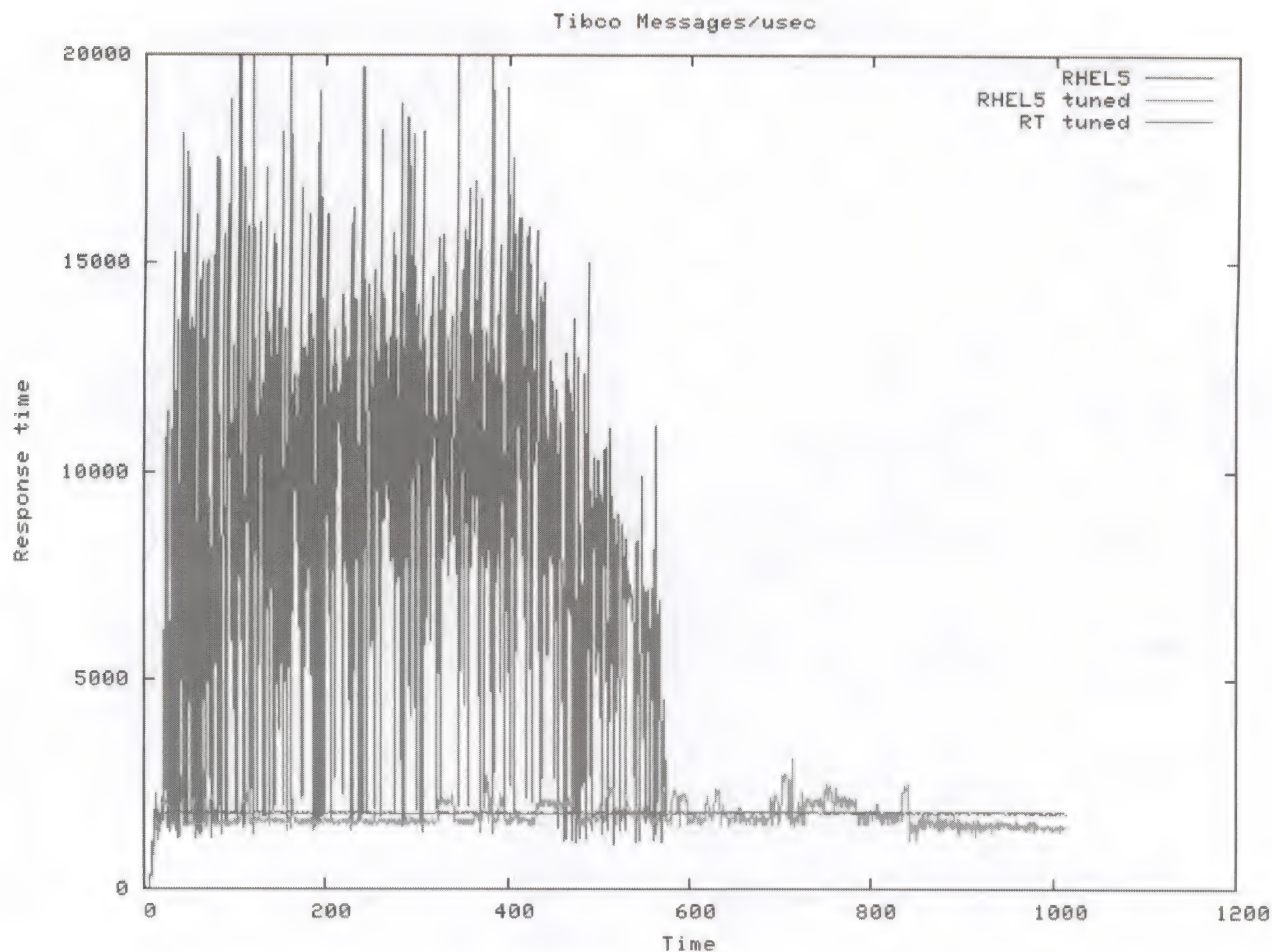


- RT Linux is an effort to make the Linux kernel deterministically responsive to events
 - The elapsed time from event occurrence to event handling is bounded
- Necessary changes for RT Linux:
 - A modified kernel that targets:
 - Fast response to events
 - Consistent response times
 - Highly precise timers
 - C library interfaces to the kernel
 - POSIX threads
 - System calls for scheduler manipulations

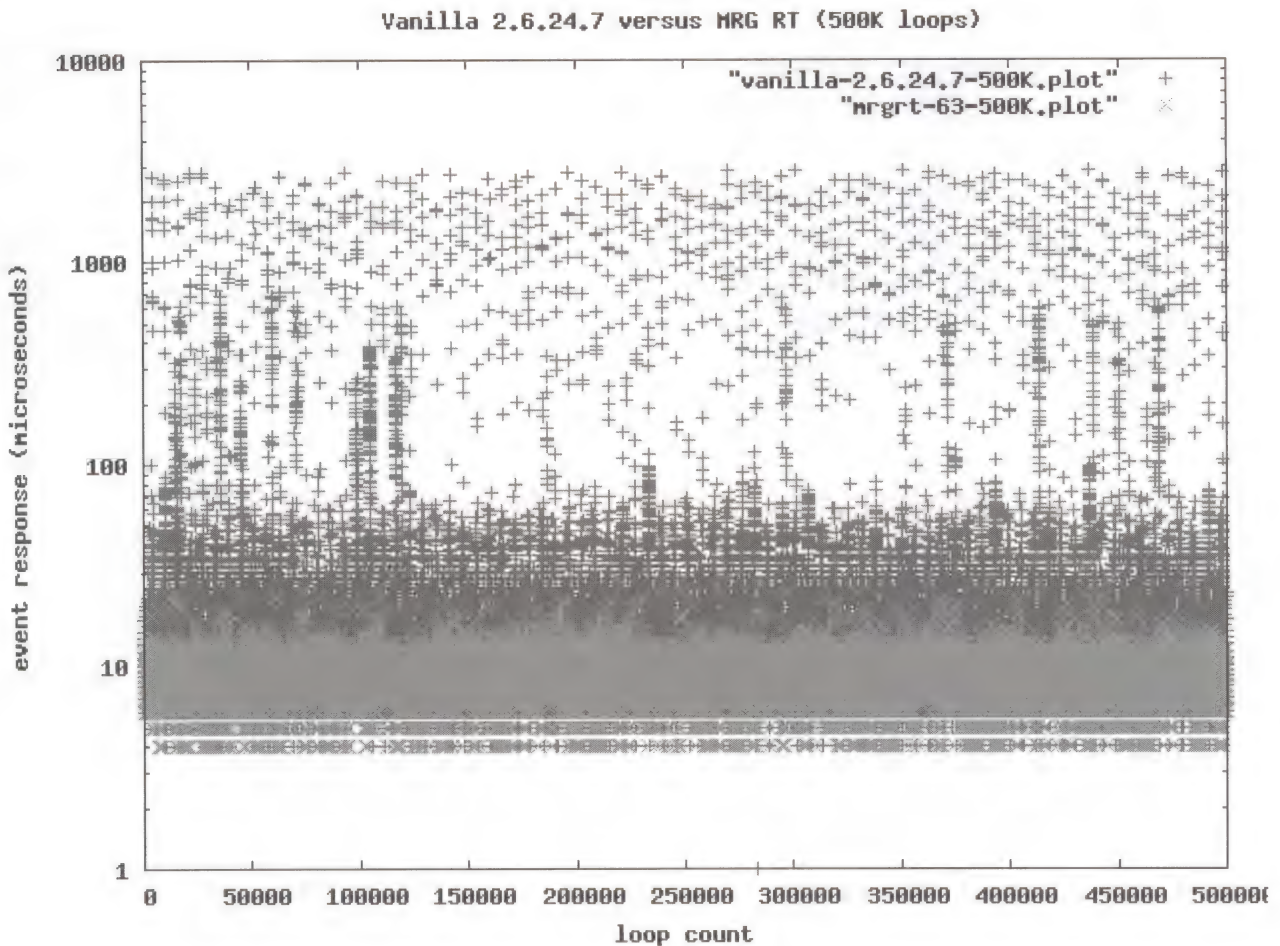


The above graphic shows the difference between running the QPID messaging application on a stock Linux kernel and an RT Linux kernel. The Y-axis is messages/s and the X-axis is of sample points (55 samples total).

Notice that the tuned RHEL5.2 kernel actually processed more messages than tuned RT kernel. Remember, the primary goal of the MRG RT kernel is determinism over raw throughput. As the plot demonstrates, the tuned RT kernel was much more consistent in its processing times. The standard deviation of messages/s on the RT kernel is much smaller than on either of the other RHEL runs (standard and tuned versions of the kernel). Again, this consistency is the biggest strength of RT.



This is another plot showing the response time between the three different kernels (stock, tuned, and RT tuned) when running the Tibco application. Notice the wildly fluctuating response times of the stock kernel. The tuned kernel calms down the response times, but relatively minor fluctuations remain. The RT kernel, in contrast, guarantees a fixed response time.



The plot above shows data from a program named `cyclictest`. The `cyclictest` program measures the difference in time from when a section of code is scheduled to wake up from when it is that it actually does wake up. The test was run on two different kernels: a vanilla 2.6.24.7 kernel and an RT kernel that is based on the same source code. A total of 500,000 loops were executed around the scheduled code, during which time the program `hackbench` was generating a load on the system in the background.

Statistically, the numbers are as follows for 500,000 samples:

	"Vanilla" kernel	RHEL RT kernel
Minimum	1	4
Maximum	2857	43
Mean	11.47	8.34
Mode	9.00	8.00
Median	9.00	8.00
Standard Deviation	54.94	1.49

Note the sharp drop in the standard deviation (a measure of the dispersion of a set of values) when running the RHEL RT version of the kernel, indicating a much greater consistency in the event response.



- Preemption
 - Most locks converted to `rt_mutex`
 - Priority inheritance for mutexes
 - threaded interrupt handlers (both hard and soft)
 - Spinlocks can sleep
 - Interrupts not turned off for almost all operations
- High-resolution timers
- Completely Fair Scheduler (CFS)
- Read-Copy-Update (RCU)
- Ftrace tracing logic



- `pthread_mutex_t` has kernel support for `PRIO_INHERIT`
 - Priority Inheritance is a mechanism used to avoid the deadlock condition known as Priority Inversion
 - The RT kernels implement priority inheritance (PI) in futexes (fast user-space mutexes) used by pthreads
- Fast user-space mutexes (futexes) used for pthread mutexes
- Timer interfaces
- Note that you don't have to have an RT kernel for most of these APIs to work
- POSIX interfaces to scheduler APIs
 - `sched_*`



- rt-tests from Thomas Gleixner
 - cyclicttest
 - signaltest
 - pi_stress
- LTP realtime tests
 - sched_latency
 - sched_football
 - pi-tests

rt.wiki.kernel.org



- **tuna**
 - Isolate processor cores
 - Adjust thread priorities
 - Change interrupt affinity
- **ftrace**
 - RT kernel built-in mechanism to trace events
 - Used to find long latency areas
- **oprofile**
 - System-wide sampling profiler
 - Used to find application hotspots
- **systemtap**
 - Custom data probes for kernel space



- Dedicate processors to your application threads
 - Use **tuna** or **taskset** to bind threads to specific processors and move other threads off
 - 4-way and 8-way processors getting cheaper
- Use cpu affinity field in `/proc/irq/n/smp_affinity` to bind interrupts to specific processors
 - **tuna** can do this easily
- Use POSIX threads
 - finer grained applications mean more parallelism, so can take advantage of multiple cores
- Use POSIX threads synchronization mechanisms
 - Mutexes
 - Barriers
 - Condition variables
- Set appropriate priorities for your threads
 - Any `SCHED_FIFO` thread is higher priority than any `SCHED_OTHER` thread
 - Ensure that you high priority threads don't hog the processor



- Red Hat Messaging, Realtime, Grid (MRG)
 - Separate product
 - Layered on Red Hat Enterprise Linux 5.2
 - Kernel is 2.6.24.7-rt11 based
 - Supports i686 and x86_64 architectures only
 - Comes with tuna and other support packages
 - No application changes required for RHEL5 applications
 - No recompiles needed



End of Lecture 17

- Questions and Answers
- Summary
 - Realtime performance characteristics

The first part of the report is a general introduction to the subject of the study. It discusses the importance of the study and the objectives of the research.

The second part of the report is a detailed description of the methodology used in the study. It includes information about the sample, the data collection methods, and the statistical analysis.

The third part of the report is a presentation of the results of the study. It includes a summary of the findings and a discussion of their implications.

The fourth part of the report is a conclusion and a list of references. The conclusion summarizes the main findings of the study and provides recommendations for future research.

The fifth part of the report is an appendix containing additional information related to the study, such as raw data, questionnaires, and interview transcripts.

The sixth part of the report is a bibliography listing the sources of information used in the study.

The seventh part of the report is a list of figures and tables, which are used to present the results of the study in a clear and concise manner.

The eighth part of the report is a list of abbreviations and acronyms used throughout the report.

The ninth part of the report is a list of symbols and units used in the study.

The tenth part of the report is a list of acknowledgments, thanking the individuals and organizations that provided support and assistance during the study.



redhat.

1-866-626-2994

<http://www.redhat.com/training>